Getting Started with Pyparsing

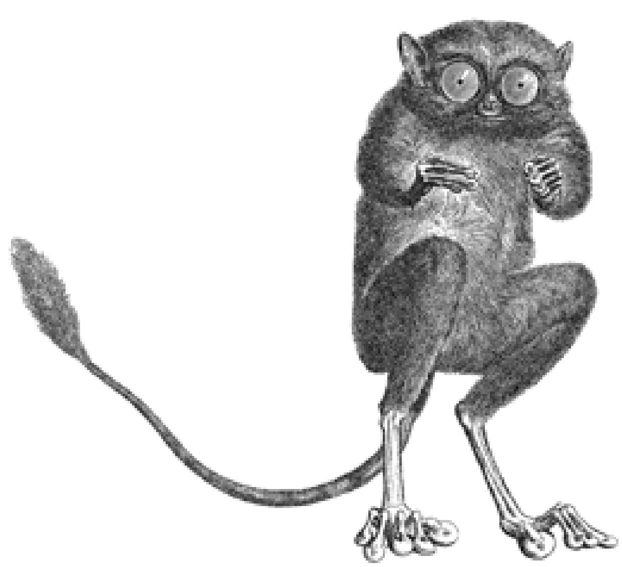
by Paul McGuire

ISBN: 9780596514235
Released: October 4, 2007

Need to extract data from a text file or a web page? Or do you want to make your application more flexible with user-defined commands or search strings? Do regular expressions and lex/yacc make your eyes blur and your brain hurt?

Pyparsing could be the solution. Pyparsing is a pure-Python class library that makes it easy to build recursive-descent parsers quickly. There is no need to handcraft your own parsing state machine. With pyparsing, you can quickly create HTML page scrapers, logfile data extractors, or complex data structure or command processors. This Short Cut shows you how!

Contents



"I need to analyze this logfile..."
"Just extract the data from this web page..."
"We need a simple input command processor..."
"Our source code needs to be migrated to the new API..."

Each of these everyday requests generates the same reflex response in any developer faced with them: "Oh, *&#$*!, not another parser!"

The task of parsing data from loosely formatted text crops up in different forms on a regular basis for most developers. Sometimes it is for one-off development utilities, such as the API-upgrade example, that are purely for internal use. Other times, the parsing task is a user-interface function to be built in to a command-driven application.

If you are working in Python, you can tackle many of these jobs using Python's built-in string methods, most notably `split()`, `index()`, and `startswith()`.

What makes parser writing unpleasant are those jobs that go beyond simple string splitting and indexing, with some context-varying format, or a structure defined to a language syntax rather than a simple character pattern. For instance,

```
y = 2 * x + 10
```

is easy to parse when split on separating spaces. Unfortunately, few users are so careful in their spacing, and an arithmetic expression parser is just as likely to encounter any of these:

```
y = 2*x + 10
y = 2*x+10
y=2*x+10
```

Splitting this last expression on whitespace just returns the original string, with no further insight into the separate elements `y`, `=`, `2`, etc.

The traditional tools for developing parsing utilities that are beyond processing with just `str.split` are *regular expressions* and *lex/yacc*. Regular expressions use a text string to describe a text pattern to be matched. The text string uses special characters (such as `|`, `+`, `.`, `*`, and `?`) to denote various parsing concepts such as alternation, repetition, and wildcards. Lex and yacc are utilities that lexically detect token boundaries, and then apply processing code to the extracted tokens. Lex and yacc use a separate token-definition file, and then generate lexing and token-processing code templates for the programmer to extend with application-specific behavior.

> **Historical note**
>
> These technologies were originally developed as text-processing and compiler generation utilities in C in the 1970s, and they continue to be in wide use today. The Python distribution includes regular expression support with the `re` module, part of its "batteries included" standard library. You can download a number of freely available parsing modules that perform lex/yacc-style parsing ported to Python.

The problem in using these traditional tools is that each introduces its own specialized notation, which must then be mapped to a Python design and Python code. In the case of lex/yacc-style tools, a separate code-generation step is usually required.

In practice, parser writing often takes the form of a seemingly endless cycle: write code, run parser on sample text, encounter unexpected text input case, modify code, rerun modified parser, find additional "special" case, etc. Combined with the notation issues of regular expressions, or the extra code-generation steps of lex/yacc, this cyclic process can spiral into frustration.

What Is Pyparsing?

Pyparsing is a pure Python module that you can add to your Python application with little difficulty. Pyparsing's class library provides a set of classes for building up a parser from individual expression elements, up to complex, variable-syntax expressions. Expressions are combined using intuitive operators, such as `+` for sequentially adding one expression after another, and `|` and `^` for defining parsing alternatives (meaning "match first alternative" or "match longest alternative"). Replication of expressions is added using classes such as `OneOrMore`, `ZeroOrMore`, and `Optional`.

For example, a regular expression that would parse an IP address followed by a U.S.-style phone number might look like the following:

```
(\d{1,3}(?:\.\d{1,3}){3})\s+(\(\d{3}\)\d{3}-\d{4})
```

In contrast, the same expression using pyparsing might be written as follows:

```
ipField = Word(nums, max=3)
ipAddr = Combine( ipField + "." + ipField + "." + ipField + "." + ipField )
phoneNum = Combine( "(" + Word(nums, exact=3) + ")" +
                    Word(nums, exact=3) + "-" + Word(nums, exact=4) )
userdata = ipAddr + phoneNum
```

Although it is more verbose, the pyparsing version is clearly more readable; it would be much easier to go back and update this version to handle international phone numbers, for example.

New to Python?

I have gotten many emails from people who were writing a pyparsing application for their first Python program. They found pyparsing to be easy to pick up, usually by adapting one of the example scripts that is included with pyparsing. If you are just getting started with Python, you may feel a bit lost going through some of the examples. Pyparsing does not require much advanced Python knowledge, so it is easy to come up to speed quickly. There are a number of online tutorial resources, starting with the Python web site, **www.python.org** [***http://www.python.org***].

To make the best use of pyparsing, you should become familiar with the basic Python language features of indented syntax, data types, and `for item in itemSequence:` control structures.

Pyparsing makes use of `object.attribute` notation, as well as Python's built-in container classes, *tuple*, *list*, and *dict*.

The examples in this book use Python *lambda*s, which are essentially one-line functions; lambdas are especially useful when defining simple parse actions.

The *list comprehension* and *generator expression* forms of iteration are useful when extracting tokens from parsed results, but not required.

Pyparsing is:

- 100 percent pure Python—no compiled *dynamic link libraries* (DLLs) or shared libraries are used in pyparsing, so you can use it on any platform that is Python 2.3-compatible.
- Driven by parsing expressions that are coded inline, using standard Python class notation and constructs —no separate code generation process and no specialized character notation make your application easier to develop, understand, and maintain.

- Enhanced with helpers for common parsing patterns:
  - C, C++, Java, Python, and HTML comments
  - quoted strings (using single or double quotes, with `\'` or `\"` escapes)
  - HTML and XML tags (including upper-/lowercase and tag attribute handling)
  - comma-separated values and delimited lists of arbitrary expressions
- Small in footprint—Pyparsing's code is contained within a single Python source file, easily dropped into a site-packages directory, or included with your own application.
- Liberally licensed—MIT license permits any use, commercial or non-commercial.

Basic Form of a Pyparsing Program

The prototypical pyparsing program has the following structure:

- Import names from pyparsing module
- Define grammar using pyparsing classes and helper methods
- Use the grammar to parse the input text
- Process the results from parsing the input text

Import Names from Pyparsing

In general, using the form `from pyparsing import *` is discouraged among Python style experts. It pollutes the local variable namespace with an unknown number of new names from the imported module. However, during pyparsing grammar development, it is hard to anticipate all of the parser element types and other pyparsing-defined names that will be needed, and this form simplifies early grammar development. After the grammar is mostly finished, you can go back to this statement and replace the * with the list of pyparsing names that you actually used.

Define the Grammar

The grammar is your definition of the text pattern that you want to extract from the input text. With pyparsing, the grammar takes the form of one or more Python statements that define text patterns, and combinations of patterns, using pyparsing classes and helpers to specify these individual pieces. Pyparsing allows you to use operators such as `+`, `|`, and `^` to simplify this code. For instance, if I use the pyparsing `Word` class to define a typical programming variable name consisting of a leading

alphabetic character with a body of alphanumeric characters or underscores, I would start with the Python statement:

```
identifier = Word(alphas, alphanums+'_')
```

I might also want to parse numeric constants, either integer or floating point. A simplistic definition uses another `Word` instance, defining our number as a "word" composed of numeric digits, possibly including a decimal point:

```
number = Word(nums+".")
```

From here, I could then define a simple assignment statement as:

```
assignmentExpr = identifier + "=" + (identifier | number)
```

and now I have a grammar that will parse any of the following:

```
a = 10
a_2=100
pi=3.14159
goldenRatio = 1.61803
E = mc2
```

In this part of the program you can also attach any parse-time callbacks (or *parse actions*) or define names for significant parts of the grammar to ease the job of locating those parts later. Parse actions are a very powerful feature of pyparsing, and I will also cover them later in detail.

Best Practice: Start with a BNF

Before just diving in and writing a bunch of stream-of-consciousness Python code to represent your grammar, take a moment to put down on paper *a description of the problem*. Having this will:

- Help clarify your thoughts on the problem
- Guide your parser design
- Give you a checklist of things to do as you implement your parser
- Help you know when you are done

Fortunately, in developing parsers, there is a simple notation to use to describe the layout for a parser called *Backus-Naur Form* (BNF). You can find good examples of BNF at ***http://en.wikipedia.org/wiki/backus-naur_form***. It is not vital that you be absolutely rigorous in your BNF notation; just get a clear idea ahead of time of what your grammar needs to include.

For the BNFs we write in this book, we'll just use this abbreviated notation:

- `::=` means "is defined as"
- + means "1 or more"
- * means "0 or more"
- items enclosed in []are optional
- succession of items means that matching tokens must occur in sequence
- | means either item may occur

Use the Grammar to Parse the Input Text

In early versions of pyparsing, this step was limited to using the `parseString` method, as in:

```
assignmentTokens = assignmentExpr.parseString("pi=3.14159")
```

to retrieve the matching tokens as parsed from the input text.

The options for using your pyparsing grammar have increased since the early versions. With later releases of pyparsing, you can use any of the following:

`parseString`

Applies the grammar to the given input text.

`scanString`

Scans through the input text looking for matches; `scanString` is a generator function that returns the matched tokens, and the start and end location within the text, as each match is found.

`searchString`

A simple wrapper around `scanString`, returning a list containing each set of matched tokens within its own sublist.

`transformString`

Another wrapper around `scanString`, to simplify replacing matched tokens with modified text or replacement strings, or to strip text that matches the grammar.

For now, let's stick with `parseString`, and I'll show you the other choices in more detail later.

Process the Results from Parsing the Input Text

Of course, the whole point of using the parser in the first place is to extract data from the input text. Many parsing tools simply return a list of the matched tokens to be further processed to interpret the meaning of the separate tokens. Pyparsing offers a rich object for results, called `ParseResults`. In its simplest form, `ParseResults` can be printed and accessed just like a Python list. For instance, continuing our assignment expression example, the following code:

```
assignmentTokens = assignmentExpr.parseString("pi=3.14159")
print assignmentTokens
```

prints out:

```
['pi', '=', '3.14159']
```

But `ParseResults` can also support direct access to individual fields within the parsed text, if results names were assigned as part of the grammar definition. By enhancing the definition of `assignmentExpr` to use results names (such as `lhs` and `rhs` for the left- and righthand sides of the assignment), we can access the fields as if they were attributes of the returned `ParseResults`:

```
assignmentExpr = identifier.setResultsName("lhs") + "=" + \
                                        (identifier | number).setResultsName("rhs")
assignmentTokens = assignmentExpr.parseString( "pi=3.14159" )
print assignmentTokens.rhs, "is assigned to", assignmentTokens.lhs
```

prints out:

```
3.14159 is assigned to pi
```

Now that the introductions are out of the way, let's move on to some detailed examples.

"Hello, World!" on Steroids!

Pyparsing comes with a number of examples, including a basic "Hello, World!" parser*. This simple example is also covered in the O'Reilly **ONLamp.com** [***http://onlamp.com***] article "Building Recursive Descent Parsers with Python" (***http://www.onlamp.com/-pub/a/python/2006/01/26/pyparsing.html***). In this section, I use this same example to introduce many of the basic parsing tools in pyparsing.

The current "Hello, World!" parsers are limited to greetings of the form:

```
word, word !
```

This limits our options a bit, so let's expand the grammar to handle more complicated greetings. Let's say we want to parse any of the following:

```
Hello, World!
Hi, Mom!
Good morning, Miss Crabtree!
Yo, Adrian!
Whattup, G?
How's it goin', Dude?
Hey, Jude!
Goodbye, Mr. Chips!
```

The first step in writing a parser for these strings is to identify the pattern that they all follow. Following our best practice, we write up this pattern as a BNF. Using ordinary words to describe a greeting, we would say, "a greeting is made up of one or more words (which is the salutation), followed by a comma, followed by one or more additional words (which is the subject of the greeting, or greetee), and ending with either an exclamation point or a question mark." As BNF, this description looks like:

```
greeting ::= salutation comma greetee endpunc
salutation ::= word+
comma ::= ,
greetee ::= word+
word ::= a collection of one or more characters, which are any alpha or ' or .
endpunc ::= ! | ?
```

* Of course, writing a parser to extract the components from "Hello, World!" is beyond overkill. But hopefully, by expanding this example to implement a generalized greeting parser, I cover most of the pyparsing basics.

This BNF translates almost directly into pyparsing, using the basic pyparsing elements `Word`, `Literal`, `OneOrMore`, and the helper method `oneOf`. (One of the translation issues in going from BNF to pyparsing is that BNF is traditionally a "top-down" definition of a grammar. Pyparsing must build its grammar "bottom-up," to ensure that referenced variables are defined before they are used.)

```
word = Word(alphas+"'.")
salutation = OneOrMore(word)
comma = Literal(",")
greetee = OneOrMore(word)
endpunc = oneOf("! ?")
greeting = salutation + comma + greetee + endpunc
```

`oneOf` is a handy shortcut for defining a list of literal alternatives. It is simpler to write:

```
endpunc = oneOf("! ?")
```

than:

```
endpunc = Literal("!") | Literal("?")
```

You can call `oneOf` with a list of items, or with a single string of items separated by whitespace.

Using our greeting parser on the set of sample strings gives the following results:

```
['Hello', ',', 'World', '!']
['Hi', ',', 'Mom', '!']
['Good', 'morning', ',', 'Miss', 'Crabtree', '!']
['Yo', ',', 'Adrian', '!']
['Whattup', ',', 'G', '?']
["How's", 'it', "goin'", ',', 'Dude', '?']
['Hey', ',', 'Jude', '!']
['Goodbye', ',', 'Mr.', 'Chips', '!']
```

Everything parses into individual tokens all right, but there is very little structure to the results. With this parser, there is quite a bit of work still to do to pick out the significant parts of each greeting string. For instance, to identify the tokens that compose the initial part of the greeting—the salutation—we need to iterate over the results until we reach the comma token:

```
for t in tests:
    results = greeting.parseString(t)
    salutation = []
    for token in results:
        if token == ",": break
        salutation.append(token)
    print salutation
```

Yuck! We might as well just have written a character-by-character scanner in the first place! Fortunately, we can avoid this drudgery by making our parser a bit smarter.

Since we know that the `salutation` and `greetee` parts of the greeting are logical groups, we can use pyparsing's `Group` class to give more structure to the returned results. By changing the definitions of `salutation` and `greetee` to:

```
salutation = Group( OneOrMore(word) )
greetee = Group( OneOrMore(word) )
```

our results start to look a bit more organized:

```
[['Hello'], ',', ['World'], '!']
[['Hi'], ',', ['Mom'], '!']
[['Good', 'morning'], ',', ['Miss', 'Crabtree'], '!']
[['Yo'], ',', ['Adrian'], '!']
[['Whattup'], ',', ['G'], '?']
[["How's", 'it', "goin'"], ',', ['Dude'], '?']
[['Hey'], ',', ['Jude'], '!']
[['Goodbye'], ',', ['Mr.', 'Chips'], '!']
```

and we can use basic list-to-variable assignment to access the different parts:

```
for t in tests:
    salutation, dummy, greetee, endpunc = greeting.parseString(t)
    print salutation, greetee, endpunc
```

prints:

```
['Hello'] ['World'] !
['Hi'] ['Mom'] !
['Good', 'morning'] ['Miss', 'Crabtree'] !
['Yo'] ['Adrian'] !
['Whattup'] ['G'] ?
["How's", 'it', "goin'"] ['Dude'] ?
['Hey'] ['Jude'] !
['Goodbye'] ['Mr.', 'Chips'] !
```

Note that we had to put in the scratch variable `dummy` to handle the parsed comma character. The comma is a very important element during parsing, since it shows where the parser stops reading the `salutation` and starts the `greetee`. But in the returned results, the comma is not really very interesting at all, and it would be nice to suppress it from the returned results. You can do this by wrapping the definition of comma in a pyparsing `Suppress` instance:

```
comma = Suppress( Literal(",") )
```

There are actually a number of shortcuts built into pyparsing, and since this function is so common, any of the following forms accomplish the same thing:

```
comma = Suppress( Literal(",") )
comma = Literal(",").suppress()
comma = Suppress(",")
```

Using one of these forms to suppress the parsed comma, our results are further cleaned up to read:

```
[['Hello'], ['World'], '!']
[['Hi'], ['Mom'], '!']
[['Good', 'morning'], ['Miss', 'Crabtree'], '!']
[['Yo'], ['Adrian'], '!']
[['Whattup'], ['G'], '?']
[["How's", 'it', "goin'"], ['Dude'], '?']
[['Hey'], ['Jude'], '!']
[['Goodbye'], ['Mr.', 'Chips'], '!']
```

The results-handling code can now drop the distracting dummy variable, and just use:

```
for t in tests:
    salutation, greetee, endpunc = greeting.parseString(t)
```

Now that we have a decent parser and a good way to get out the results, we can start to have fun with the test data. First, let's accumulate the salutations and greetees into lists of their own:

```
salutes = []
greetees = []
for t in tests:
    salutation, greetee, endpunc = greeting.parseString(t)
    salutes.append( ( " ".join(salutation), endpunc) )
    greetees.append( " ".join(greetee) )
```

I've also made a few other changes to the parsed tokens:

- Used `" ".join(list)` to convert the grouped tokens back into simple strings
- Saved the end punctuation in a tuple with each greeting to distinguish the exclamations from the questions

Now that we have collected these assorted names and salutations, we can use them to contrive some additional, never-before-seen greetings and introductions.

After importing the random module, we can synthesize some new greetings:

```
for i in range(50):
    salute = random.choice( salutes )
    greetee = random.choice( greetees )
    print "%s, %s%s" % ( salute[0], greetee, salute[1] )
```

Now we see the all-new set of greetings:

```
Hello, Miss Crabtree!
How's it goin', G?
Yo, Mr. Chips!
Whattup, World?
Good morning, Mr. Chips!
Goodbye, Jude!
Good morning, Miss Crabtree!
Hello, G!
Hey, Dude!
How's it goin', World?
Good morning, Mom!
How's it goin', Adrian?
Yo, G!
Hey, Adrian!
Hi, Mom!
Hello, Mr. Chips!
Hey, G!
Whattup, Mr. Chips?
Whattup, Miss Crabtree?
...
```

We can also simulate some introductions with the following code:

```
for i in range(50):
    print '%s, say "%s" to %s.' % ( random.choice( greetees ),
                                    "".join( random.choice( salutes ) ),
                                    random.choice( greetees ) )
```

And now the cocktail party starts shifting into high gear!

```
Jude, say "Good morning!" to Mom.
G, say "Yo!" to Miss Crabtree.
Jude, say "Goodbye!" to World.
Adrian, say "Whattup?" to World.
Mom, say "Hello!" to Dude.
Mr. Chips, say "Good morning!" to Miss Crabtree.
Miss Crabtree, say "Hi!" to Adrian.
Adrian, say "Hey!" to Mr. Chips.
Mr. Chips, say "How's it goin'?" to Mom.
G, say "Whattup?" to Mom.
Dude, say "Hello!" to World.
Miss Crabtree, say "Goodbye!" to Miss Crabtree.
Dude, say "Hi!" to Mr. Chips.
G, say "Yo!" to Mr. Chips.
World, say "Hey!" to Mr. Chips.
G, say "Hey!" to Adrian.
Adrian, say "Good morning!" to G.
Adrian, say "Hello!" to Mom.
World, say "Good morning!" to Miss Crabtree.
Miss Crabtree, say "Yo!" to G.
...
```

So, now we've had some fun with the pyparsing module. Using some of the simpler pyparsing classes and methods, we're ready to say "Whattup" to the world!

What Makes Pyparsing So Special?

Pyparsing was designed with some specific goals in mind. These goals are based on the premise that the grammar must be easy to write, to understand, and to adapt as the parsing demands on a given parser change and expand over time. The intent behind these goals is to simplify the parser design task as much as possible and to allow the pyparsing user to focus his attention on the parser, and not to be distracted by the mechanics of the parsing library or grammar syntax. The rest of this section lists the high points of the Zen of Pyparsing.

The grammar specification should be a natural-looking part of the Python program, easy-to-read, and familiar in style and format to Python programmers

Pyparsing accomplishes this in a couple of ways:

- Using operators to join parser elements together. Python's support for defining operator functions allows us to go beyond standard object construction syntax, and we can compose parsing expressions that read naturally. Instead of this:

  ```
  streetAddress = And( [streetNumber, name,
                            Or( [Literal("Rd."), Literal("St.")] ) ] )
  ```

  we can write this:

  ```
  streetAddress = streetNumber + name + ( Literal("Rd.") | Literal("St.") )
  ```

- Many attribute setting methods in pyparsing return self so that several of these methods can be chained together. This permits parser elements within a grammar to be more self-contained. For example, a common parser expression is the definition of an integer, including the specification of its name, and the attachment of a parse action to convert the integer string to a Python int. Using properties, this would look like:

  ```
  integer = Word(nums)
  integer.Name = "integer"
  integer.ParseAction = lambda t: int(t[0])
  ```

  Using attribute setters that return self, this can be collapsed to:

  ```
  integer = Word(nums).setName("integer").setParseAction(lambda t:int(t[0]))
  ```

Class names are easier to read and understand than specialized typography

This is probably the most explicit distinction of pyparsing from regular expressions, and regular expression-based parsing tools. The IP address and phone

number example given in the introduction allude to this idea, but regular expressions get truly inscrutable when regular expression control characters are also part of the text to be matched. The result is a mish-mash of backslashes to escape the control characters to be interpreted as input text. Here is a regular expression to match a simplified C function call, constrained to accept zero or more arguments that are either words or integers:

```
(\w+)\((((\d+|\w+)(,(\d+|\w+))*)?)\)
```

It is not easy to tell at a glance which parentheses are grouping operators, and which are part of the expression to be matched. Things get even more complicated if the input text contains `\`, `.`, `*`, or `?` characters. The pyparsing version of this same expression is:

```
Word(alphas)+ "(" + Group( Optional(Word(nums)|Word(alphas) +
                                     ZeroOrMore("," + Word(nums)|Word
(alphas))) ) + ")"
```

In the pyparsing version, the grouping and repetition are explicit and easy to read. In fact, this pattern of `x + ZeroOrMore(","+x)` is so common, there is a pyparsing helper method, `delimitedList`, that emits this expression. Using `delimitedList`, our pyparsing rendition further simplifies to:

```
Word(alphas)+ "(" + Group( Optional(delimitedList(Word(nums)|Word(alphas))) ) + ")"
```

Whitespace markers clutter and distract from the grammar definition

In addition to the "special characters aren't really that special" problem, regular expressions must also explicitly indicate where whitespace can occur in the input text. In this C function example, the regular expression would match:

```
abc(1,2,def,5)
```

but would not match:

```
abc(1, 2, def, 5)
```

Unfortunately, it is not easy to predict where optional whitespace might or might not occur in such an expression, so one must include `\s*` expressions liberally throughout, further obscuring the real text matching that was intended:

```
(\w+)\s*\(\s*(((\d+|\w+)(\s*,\s*(\d+|\w+))*)?)\s*\)
```

In contrast, pyparsing skips over whitespace between parser elements by default, so that this same pyparsing expression:

```
Word(alphas)+ "(" + Group( Optional(delimitedList(Word(nums)|Word(alphas))) ) + ")"
```

matches either of the listed calls to the `abc` function, without any additional whitespace indicators.

This same concept also applies to comments, which can appear anywhere in a source program. Imagine trying to match a function in which the developer had inserted a comment to document each parameter in the argument list. With pyparsing, this is accomplished with the code:

```
cFunction = Word(alphas)+ "(" + \
                    Group( Optional(delimitedList(Word(nums)|Word(alphas))) ) + ")"
cFunction.ignore( cStyleComment )
```

The results of the parsing process should do more than just represent a nested list of tokens, especially when grammars get complicated

Pyparsing returns the results of the parsing process using a class named `ParseResults`. `ParseResults` will support simple list-based access (such as indexing using [], `len`, `iter`, and slicing) for simple grammars, but it can also represent nested results, and dict-style and object attribute-style access to named fields within the results. The results from parsing our C function example are:

```
['abc', '(', ['1', '2', 'def', '5'], ')']
```

You can see that the function arguments have been collected into their own sublist, making the extraction of the function arguments easier during post-parsing analysis. If the grammar definition includes results names, specific fields can be accessed by name instead of by error-prone list indexing.

These higher-level access techniques are crucial to making sense of the results from a complex grammar.

Parse time is a good time for additional text processing

While parsing, the parser is performing many checks on the format of fields within the input text: testing for the validity of a numeric string, or matching a pattern of punctuation such as a string within quotation marks. If left as strings, the post-parsing code will have to re-examine these fields to convert them into Python ints and strings, and likely have to repeat the same validation tests before doing the conversion.

Pyparsing supports the definition of parse-time callbacks (called *parse actions*) that you can attach to individual expressions within the grammar. Since the parser calls these functions immediately after matching their respective patterns, there is often little or no extra validation required. For instance, to extract the string from the body of a parsed quoted string, a simple parse action to remove the opening and closing quotation marks, such as:

```
quotedString.setParseAction( lambda t: t[0][1:-1] )
```

is sufficient. There is no need to test the leading and trailing characters to see whether they are quotation marks—the function won't be called unless they are.

Parse actions can also be used to perform additional validation checks, such as testing whether a matched word exists in a list of valid words, and raising a `ParseException` if not. Parse actions can also return a constructed list or application object, essentially compiling the input text into a series of executable or callable user objects. Parse actions can be a powerful tool when designing a parser with pyparsing.

Grammars must tolerate change, as grammar evolves or input text becomes more challenging

The death spiral of frustration that is so common when you have to write parsers is not easy to avoid. What starts out as a simple pattern-matching exercise can become progressively complex and unwieldy. The input text can contain data that doesn't quite match the pattern but is needed anyway, so the parser gets a minor patch to include the new variation. Or, a language for which the parser was written gains a new addition to the language syntax. After this happens several times, the patches begin to get in the way of the original pattern definition, and further patches get more and more difficult. When a new change occurs after a quiet period of a few months or so, reacquiring the parser knowledge takes longer than expected, and this just adds to the frustration.

Pyparsing doesn't *cure* this problem, but the grammar definition techniques and the coding style it fosters in the grammar and parser code make many of these problems simpler. Individual elements of the grammar are likely to be explicit and easy to find, and correspondingly easy to extend or modify. Here is a wonderful quote sent to me by a pyparsing user considering writing a grammar for a particularly tricky parser: "I could just write a custom method, but my past experience was that once I got the basic pyparsing grammar working, it turned out to be more self documenting and easier to maintain/extend."

Parsing Data from a Table—Using Parse Actions and ParseResults

As our first example, let's look at a simple set of scores for college football games that might be given in a datafile. Each row of text gives the date of each game, followed by the college names and each school's score.

```
09/04/2004  Virginia              44   Temple              14
09/04/2004  LSU                   22   Oregon State        21
09/09/2004  Troy State            24   Missouri            14
01/02/2003  Florida State        103   University of Miami 2
```

Our BNF for this data is simple and clean:

```
digit      ::= '0'..'9'
alpha      ::= 'A'..'Z' 'a'..'z'
date       ::= digit+ '/' digit+ '/' digit+
schoolName ::= ( alpha+ )+
score      ::= digit+
schoolAndScore ::= schoolName score
gameResult ::= date schoolAndScore schoolAndScore
```

We begin building up our parser by converting these BNF definitions into pyparsing class instances. Just as we did in the extended "Hello, World!" program, we'll start by defining the basic building blocks that will later get combined to form the complete grammar:

```
# nums and alphas are already defined by pyparsing
num = Word(nums)
date = num + "/" + num + "/" + num
schoolName = OneOrMore( Word(alphas) )
```

Notice that you can compose pyparsing expressions using the + operator to combine pyparsing expressions and string literals. Using these basic elements, we can finish the grammar by combining them into larger expressions:

```
score = Word(nums)
schoolAndScore = schoolName + score
gameResult = date + schoolAndScore + schoolAndScore
```

We use the `gameResult` expression to parse the individual lines of the input text:

```
tests = """\
    09/04/2004  Virginia              44   Temple             14
    09/04/2004  LSU                   22   Oregon State       21
    09/09/2004  Troy State            24   Missouri           14
    01/02/2003  Florida State        103   University of Miami 2""".splitlines()

for test in tests:
    stats = gameResult.parseString(test)
    print stats.asList()
```

Just as we saw in the "Hello, World!" parser, we get an unstructured list of strings from this grammar:

```
['09', '/', '04', '/', '2004', 'Virginia', '44', 'Temple', '14']
['09', '/', '04', '/', '2004', 'LSU', '22', 'Oregon', 'State', '21']
['09', '/', '09', '/', '2004', 'Troy', 'State', '24', 'Missouri', '14']
['01', '/', '02', '/', '2003', 'Florida', 'State', '103', 'University', 'of',
'Miami', '2']
```

The first change we'll make is to combine the tokens returned by date into a single `MM/DD/YYYY` date string. The pyparsing `Combine` class does this for us by simply wrapping the composed expression:

```
date = Combine( num + "/" + num + "/" + num )
```

With this single change, the parsed results become:

```
['09/04/2004', 'Virginia', '44', 'Temple', '14']
['09/04/2004', 'LSU', '22', 'Oregon', 'State', '21']
['09/09/2004', 'Troy', 'State', '24', 'Missouri', '14']
['01/02/2003', 'Florida', 'State', '103', 'University', 'of', 'Miami', '2']
```

`Combine` actually performs two tasks for us. In addition to concatenating the matched tokens into a single string, it also enforces that the tokens are adjacent in the incoming text.

The next change to make will be to combine the school names, too. Because `Combine`'s default behavior requires that the tokens be adjacent, we will not use it, since some of the school names have embedded spaces. Instead we'll define a routine to be run at parse time to join and return the tokens as a single string. As mentioned previously, such routines are referred to in pyparsing as parse actions, and they can perform a variety of functions during the parsing process.

For this example, we will define a parse action that takes the parsed tokens, uses the string `join` function, and returns the joined string. This is such a simple parse action that it can be written as a Python *lambda*. The parse action gets hooked to a particular expression by calling `setParseAction`, as in:

```
schoolName.setParseAction( lambda tokens: " ".join(tokens) )
```

Another common use for parse actions is to do additional semantic validation, beyond the basic syntax matching that is defined in the expressions. For instance, the expression for `date` will accept `03023/808098/29921` as a valid date, and this is certainly not desirable. A parse action to validate the input date could use `time.strptime` to parse the time string into an actual date:

```
time.strptime(tokens[0],"%m/%d/%Y")
```

If `strptime` fails, then it will raise a `ValueError` exception. Pyparsing uses its own exception class, `ParseException`, for signaling whether an expression matched or not. Parse actions can raise their own exceptions to indicate that, even though the syntax matched, some higher-level validation failed. Our validation parse action would look like this:

```
def validateDateString(tokens):
    try
        time.strptime(tokens[0], "%m/%d/%Y")
    except ValueError,ve:
        raise ParseException("Invalid date string (%s)" % tokens[0])
date.setParseAction(validateDateString)
```

If we change the date in the first line of the input to 19/04/2004, we get the exception:

```
pyparsing.ParseException: Invalid date string (19/04/2004) (at char 0), (line:1, col:1)
```

Another modifier of the parsed results is the pyparsing `Group` class. `Group` does not change the parsed tokens; instead, it nests them within a sublist. `Group` is a useful class for providing structure to the results returned from parsing:

```
score = Word(nums)
schoolAndScore = Group( schoolName + score )
```

With grouping and joining, the parsed results are now structured into nested lists of strings:

```
['09/04/2004', ['Virginia', '44'], ['Temple', '14']]
['09/04/2004', ['LSU', '22'], ['Oregon State', '21']]
['09/09/2004', ['Troy State', '24'], ['Missouri', '14']]
['01/02/2003', ['Florida State', '103'], ['University of Miami', '2']]
```

Finally, we will add one more parse action to perform the conversion of numeric strings into actual integers. This is a very common use for parse actions, and it also shows how pyparsing can return structured data, not just nested lists of parsed strings. This parse action is also simple enough to implement as a lambda:

```
score = Word(nums).setParseAction( lambda tokens : int(tokens[0]) )
```

Once again, we can define our parse action to perform this conversion, without the need for error handling in case the argument to `int` is not a valid integer string. The only time this lambda will ever be called is with a string that matches the pyparsing expression `Word (nums)`, which guarantees that only valid numeric strings will be passed to the parse action.

Our parsed results are starting to look like real database records or objects:

```
['09/04/2004', ['Virginia', 44], ['Temple', 14]]
['09/04/2004', ['LSU', 22], ['Oregon State', 21]]
['09/09/2004', ['Troy State', 24], ['Missouri', 14]]
['01/02/2003', ['Florida State', 103], ['University of Miami', 2]]
```

At this point, the returned data is structured and converted so that we could do some actual processing on the data, such as listing the game results by date and marking the winning team. The `ParseResults` object passed back from `parseString` allows us to index into the parsed data using nested list notation, but for data with this kind of structure, things get ugly fairly quickly:

```
for test in tests:
    stats = gameResult.parseString(test)
    if stats[1][1] != stats[2][1]:
        if stats[1][1] > stats[2][1]:
```

```
            result = "won by " + stats[1][0]
        else:
            result = "won by " + stats[2][0]
    else:
        result = "tied"
    print "%s %s(%d) %s(%d), %s" % (stats[0], stats[1][0], stats[1][1],
                                              stats[2][0], stats[2][1], result)
```

Not only do the indexes make the code hard to follow (and easy to get wrong!), the processing of the parsed data is very sensitive to the order of things in the results. If our grammar included some optional fields, we would have to include other logic to test for the existence of those fields, and adjust the indexes accordingly. This makes for a *very* fragile parser.

We could try using multiple variable assignment to reduce the indexing like we did in '"Hello, World!" on Steroids!':

```
for test in tests:
    stats = gameResult.parseString(test)
    gamedate,team1,team2 = stats  # <- assign parsed bits to individual variable names
    if team1[1] != team2[1]:
        if team1[1] > team2[1]:
            result = "won by " + team1[0]
        else:
            result = "won by " + team2[0]
    else:
        result = "tied"
    print "%s %s(%d) %s(%d), %s" % (gamedate, team1[0], team1[1], team2[0], team2[1],
             result)
```

Best Practice: Use Results Names

Use results names to simplify access to specific tokens within the parsed results, and to protect your parser from later text and grammar changes, and from the variability of optional data fields.

But this still leaves us sensitive to the order of the parsed data.

Instead, we can define names in the grammar that different expressions should use to label the resulting tokens returned by those expressions. To do this, we insert calls to `setResults-Name` into our grammar, so that expressions will label the tokens as they are accumulated into the `Parse-Results` for the overall grammar:

```
schoolAndScore = Group(
schoolName.setResultsName("school") +
       score.setResultsName("score") )
```

```
gameResult = date.setResultsName("date") + schoolAndScore.setResultsName("team1") +
      schoolAndScore.setResultsName("team2")
```

And the code to process the results is more readable:

```
    if stats.team1.score != stats.team2.score
        if stats.team1.score > stats.team2.score:
            result = "won by " + stats.team1.school
        else:
            result = "won by " + stats.team2.school
    else:
        result = "tied"
    print "%s %s(%d) %s(%d), %s" % (stats.date, stats.team1.school, stats.team1.
score,
             stats.team2.school, stats.team2.score, result)
```

This code has the added bonus of being able to refer to individual tokens by name rather than by index, making the processing code immune to changes in the token order and to the presence/absence of optional data fields.

Creating `ParseResults` with results names will enable you to use dict-style semantics to access the tokens. For example, you can use `ParseResults` objects to supply data values to interpolated strings with labeled fields, further simplifying the output code:

```
print "%(date)s %(team1)s %(team2)s" % stats
```

This gives the following:

```
09/04/2004 ['Virginia', 44] ['Temple', 14]
09/04/2004 ['LSU', 22] ['Oregon State', 21]
09/09/2004 ['Troy State', 24] ['Missouri', 14]
01/02/2003 ['Florida State', 103] ['University of Miami', 2]
```

`ParseResults` also implements the `keys()`, `items()`, and `values()` methods, and supports key testing with Python's `in` keyword.



Coming Attractions!

The latest version of Pyparsing (1.4.7) includes notation to make it even easier to add results names to expressions, reducing the grammar code in this example to:

```
schoolAndScore =
    Group( schoolName("school") +
            score("score") )
gameResult = date("date") +
            schoolAndScore("team1") +
            schoolAndScore("team2")
```

Now there is no excuse for not naming your parsed results!

For debugging, you can call `dump()` to return a string showing the nested token list, followed by a hierarchical listing of keys and values. Here is a sample of calling `stats.dump()` for the first line of input text:

```
print stats.dump()

['09/04/2004', ['Virginia', 44],
['Temple', 14]]
- date: 09/04/2004
- team1: ['Virginia', 44]
    - school: Virginia
    - score: 44
- team2: ['Temple', 14]
    - school: Temple
    - score: 14
```

Finally, you can generate XML representing this same hierarchy by calling `stats.asXML()` and specifying a root element name:

```
print stats.asXML("GAME")

<GAME>
    <date>09/04/2004</date>
    <team1>
        <school>Virginia</school>
        <score>44</score>
    </team1>
```

```
    <team2>
        <school>Temple</school>
        <score>14</score>
    </team2>
</GAME>
```

There is one last issue to deal with, having to do with validation of the input text. Pyparsing will parse a grammar until it reaches the end of the grammar, and then return the matched results, even if the input string has more text in it. For instance, this statement:

```
word = Word("A")
data = "AAA AA AAA BA AAA"
print OneOrMore(word).parseString(data)
```

will not raise an exception, but simply return:

```
['AAA', 'AA', 'AAA']
```

Helpful Tip – end your grammar with stringEnd

Make sure there is no dangling input text by ending your grammar with `stringEnd`, or by appending `stringEnd` to the grammar used to call `parseString`. If the grammar does not match all of the given input text, it will raise a `ParseException` at the point where the parser stopped parsing.

even though the string continues with more "AAA" words to be parsed. Many times, this "extra" text is really more data, but with some mismatch that does not satisfy the continued parsing of the grammar.

To check whether your grammar has processed the entire string, pyparsing provides a class `StringEnd` (and a built-in expression `stringEnd`) that you can add to the end of the grammar. This is your way of signifying, "at this point, I expect there to be no more text—this should be the end of the input string." If the grammar has left some part of the input unparsed, then `StringEnd` will raise a `ParseException`. Note that if there is trailing whitespace, pyparsing will automatically skip over it before testing for end-of-string.

In our current application, adding `stringEnd` to the end of our parsing expression will protect against accidentally matching

```
09/04/2004  LSU                   2x2   Oregon State      21
```

as:

```
09/04/2004 ['LSU', 2] ['x', 2]
```

treating this as a tie game between LSU and College X. Instead we get a `ParseException` that looks like:

```
pyparsing.ParseException: Expected stringEnd (at char 44), (line:1, col:45)
```

Here is a complete listing of the parser code:

```
from pyparsing import Word, Group, Combine, Suppress, OneOrMore, alphas, nums,\
   alphanums, stringEnd, ParseException
import time
num = Word(nums)
date = Combine(num + "/" + num + "/" + num)

def validateDateString(tokens):
    try:
        time.strptime(tokens[0], "%m/%d/%Y")
    except ValueError,ve:
        raise ParseException("Invalid date string (%s)" % tokens[0])
date.setParseAction(validateDateString)

schoolName = OneOrMore( Word(alphas) )
schoolName.setParseAction( lambda tokens: " ".join(tokens) )
score = Word(nums).setParseAction(lambda tokens: int(tokens[0]))
schoolAndScore = Group( schoolName.setResultsName("school") + \
        score.setResultsName("score") )
gameResult = date.setResultsName("date") + schoolAndScore.setResultsName("team1") + \
        schoolAndScore.setResultsName("team2")

tests = """\
    09/04/2004  Virginia              44   Temple              14
    09/04/2004  LSU                   22   Oregon State        21
    09/09/2004  Troy State            24   Missouri            14
    01/02/2003  Florida State        103   University of Miami 2""".splitlines()

for test in tests:
    stats = (gameResult + stringEnd).parseString(test)

    if stats.team1.score != stats.team2.score:
        if stats.team1.score > stats.team2.score:
            result = "won by " + stats.team1.school
        else:
            result = "won by " + stats.team2.school
    else:
        result = "tied"
    print "%s %s(%d) %s(%d), %s" % (stats.date, stats.team1.school, stats.team1.score,
            stats.team2.school, stats.team2.score, result)
    # or print one of these alternative formats
    #print "%(date)s %(team1)s %(team2)s" % stats
    #print stats.asXML("GAME")
```

Extracting Data from a Web Page

The Internet has become a vast source of freely available data no further than the browser window on your home computer. While some resources on the Web are formatted for easy consumption by computer programs, the majority of content is intended for human readers using a browser application, with formatting done using HTML markup tags.

Sometimes you have your own Python script that needs to use tabular or reference data from a web page. If the data has not already been converted to easily processed comma-separated values or some other digestible format, you will need to write a parser that "reads around" the HTML tags and gets the actual text data.

It is very common to see postings on Usenet from people trying to use regular expressions for this task. For instance, someone trying to extract image reference tags from a web page might try matching the tag pattern `"<img src=`*quoted_string*`>"`. Unfortunately, since HTML tags can contain many optional attributes, and since web browsers are very forgiving in processing sloppy HTML tags, HTML retrieved from the wild can be full of surprises to the unwary web page scraper. Here are some typical "gotchas" when trying to find HTML tags:

Tags with extra whitespace or of varying upper-/lowercase

`<img src="sphinx.jpeg">`, `<IMG SRC="sphinx.jpeg">`, and `<img src = "sphinx.jpeg" >` are all equivalent tags.

Tags with unexpected attributes

The `IMG` tag will often contain optional attributes, such as `align`, `alt`, `id`, `vspace`, `hspace`, `height`, `width`, etc.

Tag attributes in varying order

If the matching pattern is expanded to detect the attributes `src`, `align`, and `alt`, as in the tag `<img src="sphinx.jpeg" align="top" alt="The Great Sphinx">`, the attributes can appear in the tag in any order.

Tag attributes may or may not be enclosed in quotes

`<img src="sphinx.jpeg">` can also be represented as `<img src='sphinx.jpeg'>` or `<img src=sphinx.jpeg>`.

Pyparsing includes the helper method `makeHTMLTags` to make short work of defining standard expressions for opening and closing tags. To use this method, your program calls `makeHTMLTags` with the tag name as its argument, and `makeHTMLTags` returns pyparsing expressions for matching the opening and closing tags for the

given tag name. But `makeHTMLTags("X")` goes far beyond simply returning the expressions `Literal("<X>")` and `Literal("</X>")`:

- Tags may be upper- or lowercase.
- Whitespace may appear anywhere in the tag.
- Any number of attributes can be included, in any order.
- Attribute values can be single-quoted, double-quoted, or unquoted strings.
- Opening tags may include a terminating `/`, indicating no body text and no closing tag (specified by using the results name 'empty').
- Tag and attribute names can include namespace references.

But perhaps the most powerful feature of the expressions returned by `makeHTMLTags` is that the parsed results include the opening tag's HTML attributes named results, dynamically creating the results names while parsing.

Here is a short script that searches a web page for image references, printing a list of images and any provided alternate text:

> **Note**
>
> The standard Python library includes the modules `HTMLParser` and `htmllib` for processing HTML source, although they intolerant of HTML that is not well behaved. A popular third-party module for HTML parsing is BeautifulSoup. Here is the BeautifulSoup rendition of the `<IMG>` tag extractor:
>
> ```
> from BeautifulSoup import BeautifulSoup
>
> soup = BeautifulSoup(html)
> imgs = soup.findAll("img")
> for img in imgs:
>     print "'%(alt)s' : %(src)s" % img
> ```
>
> BeautifulSoup works by processing the entire HTML page, and provides a Pythonic hybrid of DOM and XPATH structure and data access to the parsed HTML tags, attributes, and text fields.

```
from pyparsing import makeHTMLTags
import urllib

# read data from web page
url = "https://www.cia.gov/library/"\
        "publications/the-world-"\
        "factbook/docs/refmaps.html"
html = urllib.urlopen(url).read()
```

```
# define expression for <img> tag
imgTag,endImgTag = makeHTMLTags("img")

# search for matching tags, and
# print key attributes
for img in imgTag.searchString(html):
    print "'%(alt)s' : %(src)s" % img
```

Notice that instead of using `parseString`, this script searches for matching text with `searchString`. For each match returned by `searchString`, the script prints the values of the `alt` and `src` tag attributes just as if they were attributes of the parsed tokens returned by the `img` expression.

This script just lists out images from the initial page of maps included in the online CIA Factbook. The output contains information on each map image reference, like this excerpt:

```
'Africa Map' : ../reference_maps/thumbnails/africa.jpg
'Antarctic Region Map' : ../reference_maps/thumbnails/antarctic.jpg
'Arctic Region Map' : ../reference_maps/thumbnails/arctic.jpg
'Asia Map' : ../reference_maps/thumbnails/asia.jpg
'Central America and Caribbean Map' : ../reference_maps/thumbnails/central_america.jpg
'Europe Map' : ../reference_maps/thumbnails/europe.jpg
...
```

The CIA Factbook web site also includes a more complicated web page, which lists the conversion factors for many common units of measure used around the world. Here are some sample data rows from this table:

ares	square meters	100
ares	square yards	119.599
barrels, US beer	gallons	31
barrels, US beer	liters	117.347 77
barrels, US petroleum	gallons (British)	34.97
barrels, US petroleum	gallons (US)	42
barrels, US petroleum	liters	158.987 29
barrels, US proof spirits	gallons	40
barrels, US proof spirits	liters	151.416 47
bushels (US)	bushels (British)	0.968 9
bushels (US)	cubic feet	1.244 456
bushels (US)	cubic inches	2,150.42

The corresponding HTML source for these rows is of the form:

```
<TR align="left" valign="top" bgcolor="#FFFFFF">
    <td width=33% valign=top class="Normal">ares </TD>
    <td width=33% valign=top class="Normal">square meters </TD>
    <td width=33% valign=top class="Normal">100 </TD>
  </TR>
  <TR align="left" valign="top" bgcolor="#CCCCCC">
    <td width=33% valign=top class="Normal">ares </TD>
    <td width=33% valign=top class="Normal">square yards </TD>
    <td width=33% valign=top class="Normal">119.599 </TD>
  </TR>
  <TR align="left" valign="top" bgcolor="#FFFFFF">
    <td width=33% valign=top class="Normal">barrels, US beer </TD>
    <td width=33% valign=top class="Normal">gallons </TD>
    <td width=33% valign=top class="Normal">31 </TD>
  </TR>
  <TR align="left" valign="top" bgcolor="#CCCCCC">
    <td width=33% valign=top class="Normal">barrels, US beer </TD>
    <td width=33% valign=top class="Normal">liters </TD>
    <td width=33% valign=top class="Normal">117.347 77 </TD>
  </TR>
...
```

Since we have some sample HTML to use as a template, we can create a simple BNF using shortcuts for opening and closing tags (meaning the results from `makeHTMLTags`, with the corresponding support for HTML attributes):

```
entry ::= <tr> conversionLabel conversionLabel conversionValue </tr>
conversionLabel ::= <td> text </td>
conversionValue ::= <td> readableNumber </td>
```

Note that the conversion factors are formatted for easy reading (by humans, that is):

- Integer part is comma-separated on the thousands
- Decimal part is space-separated on the thousandths

We can plan to include a parse action to reformat this text before calling `float()` to convert to a floating-point number. We will also need to post-process the text of the conversion labels; as we will find, these can contain embedded `<BR>` tags for explicit line breaks.

From a purely mechanical point of view, our script must begin by extracting the source text for the given URL. I usually find the Python `urllib` module to be sufficient for this task:

```
import urllib
url = "https://www.cia.gov/library/publications/" \
```

```
        "the-world-factbook/appendix/appendix-g.html"
html = urllib.urlopen(url).read()
```

At this point we have retrieved the web page's source HTML into our Python variable html as a single string. We will use this string later to scan for conversion factors.

But we've gotten a little ahead of ourselves—we need to set up our parser's grammar first! Let's start with the real numbers. Looking through this web page, there are numbers such as:

```
200
0.032 808 40
1,728
0.028 316 846 592
3,785.411 784
```

Here is an expression to match these numbers:

```
decimalNumber = Word(nums, nums+",") + Optional("." + OneOrMore(Word(nums)))
```

Notice that we are using a new form of the `Word` constructor, with two arguments instead of just one. When using this form, `Word` will use the first argument as the set of valid *starting* characters, and the second argument as the set of valid *body* characters. The given expression will match `1,000`, but not `,456`. This two-argument form of `Word` is useful when defining expressions for parsing identifiers from programming source code, such as this definition for a Python variable name:

```
Word(alphas+"_", alphanums+"_").
```

Since `decimalNumber` is a working parser all by itself, we can test it in isolation before including it into a larger, more complicated expression.

Best Practice: Incremental Testing

Test individual grammar elements to avoid surprises when merging them into the larger overall grammar.

Using the list of sampled numbers, we get these results:

```
['200']
['0', '.', '032', '808', '40']
['1,728']
['0', '.', '028', '316', '846', '592']
['3,785', '.', '411', '784']
```

In order to convert these into floating-point values, we need to:

- Join the individual token pieces together
- Strip out the commas in the integer part

While these two steps could be combined into a single expression, I want to create two parse actions to show how parse actions can be chained together.

The first parse action will be called `joinTokens`, and can be performed by a lambda:

```
joinTokens = lambda tokens : "".join(tokens)
```

The next parse action will be called `stripCommas`. Being the next parse action in the chain, `stripCommas` will receive a single string (the output of `joinTokens`), so we will only need to work with the 0th element of the supplied tokens:

```
stripCommas = lambda tokens : tokens[0].replace(",", "")
```

And of course, we need a final parse action to do the conversion to float:

```
convertToFloat = lambda tokens : float(tokens[0])
```

Now, to assign multiple parse actions to an expression, we can use the pair of methods, `setParseAction` and `addParseAction`:

```
decimalNumber.setParseAction( joinTokens )
decimalNumber.addParseAction( stripCommas )
decimalNumber.addParseAction( convertToFloat )
```

Or, we can just call `setParseAction` listing multiple parse actions as separate arguments, and these will be defined as a chain of parse actions to be executed in the same order that they are given:

```
decimalNumber.setParseAction( joinTokens, stripCommas, convertToFloat )
```

Next, let's do a more thorough test by creating the expression that uses `decimalNumber` and scanning the complete HTML source.

```
tdStart,tdEnd = makeHTMLTags("td")
conversionValue = tdStart + decimalNumber + tdEnd

for tokens,start,end in conversionValue.scanString(html):
    print tokens
```

`scanString` is another parsing method that is especially useful when testing grammar fragments. While `parseString` works only with a complete grammar, beginning with the start of the input string and working until the grammar is completely matched, `scanString` scans through the input text, looking for bits of the text that match the grammar. Also, `scanString` is a generator function, which means it will return tokens as they are found rather than parsing all of the input text, so your program begins to report matching tokens right away. From the code sample, you

can see that `scanString` returns the tokens and starting and ending locations for each match.

Here are the initial results from using `scanString` to test out the `conversionValue` expression:

```
['td', ['width', '33%'], ['valign', 'top'], ['class', 'Normal'], False,
40.468564223999998, '</td>']
['td', ['width', '33%'], ['valign', 'top'], ['class', 'Normal'], False,
0.40468564223999998, '</td>']
['td', ['width', '33%'], ['valign', 'top'], ['class', 'Normal'], False, 43560.0,
'</td>']
['td', ['width', '33%'], ['valign', 'top'], ['class', 'Normal'], False,
0.0040468564224000001, '</td>']
['td', ['width', '33%'], ['valign', 'top'], ['class', 'Normal'], False,
4046.8564224000002, '</td>']
['td', ['width', '33%'], ['valign', 'top'], ['class', 'Normal'], False,
0.0015625000000000001, '</td>']
['td', ['width', '33%'], ['valign', 'top'], ['class', 'Normal'], False, 4840.0,
'</td>']
...
```

Well, all those parsed tokens from the attributes of the `<TD>` tags are certainly distracting. We should clean things up by adding a results name to the `decimalNumber` expression and just printing out that part:

```
conversionValue = tdStart + decimalNumber.setResultsName("factor") + tdEnd

for tokens,start,end in conversionValue.scanString(html):
    print tokens.factor
```

Now our output is plainly:

```
40.468564224
0.40468564224
43560.0
0.0040468564224
4046.8564224
0.0015625
4840.0
100.0
...
```

Also note from the absence of quotation marks that these are not strings, but converted floats. On to the remaining elements!

We've developed the expression to extract the conversion factors themselves, but these are of little use without knowing the "from" and "to" units. To parse these, we'll use an expression very similar to the one for extracting the conversion factors:

```
fromUnit = tdStart + units.setResultsName("fromUnit") + tdEnd
toUnit   = tdStart + units.setResultsName("toUnit") + tdEnd
```

But how will we define the `units` expression itself? Looking through the web page, this text doesn't show much of a recognizable pattern. We could try something like `OneOrMore (Word(alphas))`, but that would fail when trying to match units of "barrels, US petroleum" or "gallons (British)." Trying to add in punctuation marks sets us up for errors when we overlook a little-used mark and unknowingly skip over a valid conversion factor.

One thing we do know is that the `units` text ends when we reach the closing `</TD>` tag. With this knowledge, we can avoid trying to exhaustively define the pattern for `units`, and use a helpful pyparsing class, `SkipTo`. `SkipTo` collects all the intervening text, from the current parsing position to the location of the target expression, into a single string. Using `SkipTo`, we can define `units` simply as:

```
units = SkipTo( tdEnd )
```

We may end up having to do some post-processing on this text, such as trimming leading or trailing whitespace, but at least we won't omit some valid units, and we won't read past any closing `</TD>` tags.

We are just about ready to complete our expression for extracting unit conversions, adding expressions for the "from" and "to" unit expressions:

```
conversion = trStart + fromUnits + toUnits + conversionValue + trEnd
```

Repeating the test scanning process, we get the following:

```
for tokens,start,end in conversion.scanString(html):
    print "%(fromUnit)s : %(toUnit)s : %(factor)f" % tokens

acres  : ares  : 40.468564
acres  : hectares  : 0.404686
acres  : square feet  : 43560.000000
acres  : square kilometers  : 0.004047
acres  : square meters  : 4046.856422
...
```

This doesn't seem too bad, but further down the list, there are some formatting problems:

```
barrels, US petroleum  : liters  : 158.987290
barrels, US proof
                     spirits  : gallons  : 40.000000
barrels, US proof
                     spirits  : liters  : 151.416470
bushels (US)  : bushels (British)  : 0.968900
```

And even further down, we find these entries:

```
tons, net register  : cubic feet of permanently enclosed space <br>
                      for cargo and passengers  : 100.000000
tons, net register  : cubic meters of permanently enclosed space <br>
                      for cargo and passengers  : 2.831685
```

A Note on Web Page Scraping

As helpful as much of this data is on the Internet, it is usually provided under certain Terms of Service (TOS) that do not always permit automated gathering of data items with HTML processing scripts. Usually these terms are intended to deter someone from taking a web site's data for the purpose of creating a competing or derivative site. But in other cases, the terms are there to ensure that the web site is used fairly by the Internet's human users, that the site's servers aren't unduly loaded down with automated page scrapers, and that the web site owner gets fairly compensated for providing the site's content. Most TOS do allow for extraction of the data for one's own personal use and reference. Always check the site's Terms of Service before just helping yourself to its data.

For the record, the content of the CIA Factbook web site is in the public domain (see ***https://www.cia.gov/about-cia/site-policies/index.html#link1***).

So, to clean up the units of measure, we need to strip out newlines and extra spaces, and remove embedded `<br>` tags. As you may have guessed, we'll use a parse action to do the job.

Our parse action has two tasks:

- Remove `<br>` tags
- Collapse whitespace and newlines

The simplest way in Python to collapse repeated whitespace is to use the `str` type's methods `split` followed by `join`. To remove the `<br>` tags, we will just use `str.replace("<br>"," ")`. A single lambda for both these tasks will get a little difficult to follow, so this time we'll create an actual Python method, and attach it to the `units` expression:

```
def htmlCleanup(t):
    unitText = t[0]
    unitText = unitText.replace("<br>"," ")
    unitText = " ".join(unitText.split())
    return unitText

units.setParseAction(htmlCleanup)
```

With these changes, our conversion factor extractor can collect the unit conversion information. We can load it into a Python dict variable or a local database for further use by our program.

Here is the complete conversion factor extraction program:

```
import urllib
from pyparsing import *

url = "https://www.cia.gov/library/" \
      "publications/the-world-factbook/" \
      "appendix/appendix-g.html"
page = urllib.urlopen(url)
html = page.read()
page.close()

tdStart,tdEnd = makeHTMLTags("td")
trStart,trEnd = makeHTMLTags("tr")
decimalNumber = Word(nums+",") + Optional("." + OneOrMore(Word(nums)))
joinTokens = lambda tokens : "".join(tokens)
stripCommas = lambda tokens: tokens[0].replace(",","")
convertToFloat = lambda tokens: float(tokens[0])
decimalNumber.setParseAction( joinTokens, stripCommas, convertToFloat )

conversionValue = tdStart + decimalNumber.setResultsName("factor") + tdEnd

units = SkipTo(tdEnd)
def htmlCleanup(t):
    unitText = t[0]
    unitText = " ".join(unitText.split())
    unitText = unitText.replace("<br>","")
    return unitText
units.setParseAction(htmlCleanup)

fromUnit = tdStart + units.setResultsName("fromUnit") + tdEnd
toUnit   = tdStart + units.setResultsName("toUnit") + tdEnd
conversion = trStart + fromUnit + toUnit + conversionValue + trEnd

for tokens,start,end in conversion.scanString(html):
    print "%(fromUnit)s : %(toUnit)s : %(factor)s" % tokens
```

A Simple S-Expression Parser

S-expressions are a plain ASCII form for representing complex data structures. They can be used to serialize data to send over a communication path, to persist into a database, or to otherwise use as a string representation of a hierarchical data element. When displayed in indented form, S-expressions are even suitable for human comprehension, providing a simple and intuitive nesting syntax, with pa-

rentheses used as the nesting operators. Here is a sample S-expression describing an authentication certificate:

```
(certificate
 (issuer
  (name
   (public-key
    rsa-with-md5
    (e |NFGq/E3wh9f4rJIQVXhS|)
    (n |d738/4ghP9rFZ0gAIYZ5q9y6iskDJwASi5rEQpEQq8ZyMZeIZzIAR2I5iGE=|))
   aid-committee))
 (subject
  (ref
   (public-key
    rsa-with-md5
    (e |NFGq/E3wh9f4rJIQVXhS|)
    (n |d738/4ghP9rFZ0gAIYZ5q9y6iskDJwASi5rEQpEQq8ZyMZeIZzIAR2I5iGE=|))
   tom
   mother))
(not-after "1998-01-01_09:00:00")
 (tag
(spend (account "12345678") (* numeric range "1" "1000"))))
```

The attraction of S-expressions is that they consist purely of lists of basic character or numeric strings, with structure represented using nested parentheses.

The languages Lisp and Scheme use S-expressions as their actual program syntax. Here is a factorial function written in Common Lisp:

```
(defun factorial (x)
   (if (zerop x) 1
       (* x (factorial (- x 1)))))
```

The online Wikipedia article (***http://en.wikipedia.org/wiki/s-expression***) has more background and additional links for further information on S-expressions.

In computer science classes, it is common to assign as homework the development of an S-expression parser. Doing so with pyparsing is actually a fairly straightforward task. This is also our first case of a *recursive* grammar, in which some expressions can be written in terms of other expressions of the same type.

Let's start with a very simple S-expression form, in which an expression can be a single alphabetic word or integer, or a sequence of alphabetic words or integers enclosed in parentheses. Following our standard practice, we start by defining the BNF for an S-expression:

```
alphaword ::= alphas+
integer   ::= nums+
sexp      ::= alphaword | integer | '(' sexp* ')'
```

The first two expressions are nothing new, and you can see how we would use pyparsing's `Word` class to define them:

```
alphaword = Word(alphas)
integer = Word(nums)
```

But `sexp` is more difficult. Our dilemma is that the definition for `sexp` includes `sexp` itself, but to define `sexp` we need to refer to `sexp`!

To resolve this "chicken-and-egg" problem, pyparsing provides the `Forward` class, which allows you to "forward" declare an expression before you can fully define it. The first step is to declare sexp as an empty `Forward`:

```
sexp = Forward()
```

At this point, `sexp` has no grammar definition yet. To assign the recursive definition of `sexp` into `sexp`, we use the `<<` shift operator:

```
LPAREN = Suppress("(")
RPAREN = Suppress(")")
sexp << ( alphaword | integer | ( LPAREN + ZeroOrMore(sexp) + RPAREN )
```

The `<<` operator "injects" the definition into the existing `sexp` variable. `sexp` now safely contains a reference to itself as part of its recursive definition.

Let's test this simple S-expression parser all by itself:

```
tests = """\
    red
    100
    ( red 100 blue )
    ( green ( ( 1 2 ) mauve ) plaid () )""".splitlines()

for t in tests:
    print t
    print sexp.parseString(t)
    print
```

This gives us the following results:

```
red
['red']

100
['100']

( red 100 blue )
['red', '100', 'blue']

( green ( 1 2 mauve ) plaid () )
['green', '1', '2', 'mauve', 'plaid']
```

This successfully parses all of the expressions, but that last test case is a little disappointing. There is no representation of the subexpressions defined within the nested parentheses. Once again, the `Group` class provides the missing link.

Remember that `Group` causes the matched tokens to be enclosed within their own sublist. By changing the definition of `sexp` to:

```
sexp << ( alphaword | integer | Group( LPAR + ZeroOrMore(sexp) + RPAR ) )
```

the elements within nested parentheses end up within a nested sublist in the results. And, since this `Group` construct is defined within the recursive part of the grammar, this nesting will work recursively as deeply nested as the parenthesis nesting in the original string. With this change, our results become:

```
red
['red']

100
['100']

( red 100 blue )
[['red', '100', 'blue']]

( green ( ( 1 2 ) mauve ) plaid () )
[['green', [['1', '2'], 'mauve'], 'plaid', []]]
```

Much better!

A Complete S-Expression Parser

As it turns out, there is a formal definition for S-expressions to handle many applications requiring the representation of hierarchical data. You can imagine that this goes beyond data represented as words and integers—the authentication certificate sample in the previous section requires a variety of different field types.

The Internet Draft describing S-expressions for data representation can be found at ***http://people.csail.mit.edu/rivest/sexp.txt***. Fortunately, the draft defines the BNF for us:

```
<sexp>         :: <string> | <list>
<string>       :: <display>? <simple-string> ;
<simple-string> :: <raw> | <token> | <base-64> | <hexadecimal> | <quoted-string> ;
<display>      :: "[" <simple-string> "]" ;
<raw>          :: <decimal> ":" <bytes> ;
<decimal>      :: <decimal-digit>+ ;
               -- decimal numbers should have no unnecessary leading zeros
<bytes>        -- any string of bytes, of the indicated length
<token>        :: <tokenchar>+ ;
<base-64>      :: <decimal>? "|" ( <base-64-char> | <whitespace> )* "|" ;
```

```
<hexadecimal> :: "#" ( <hex-digit> | <white-space> )* "#" ;
<quoted-string> :: <decimal>? <quoted-string-body>
<quoted-string-body> :: "\"" <bytes> "\""
<list>        :: "(" ( <sexp> | <whitespace> )* ")" ;
<whitespace>  :: <whitespace-char>* ;
<token-char>  :: <alpha> | <decimal-digit> | <simple-punc> ;
<alpha>       :: <upper-case> | <lower-case> | <digit> ;
<lower-case>  :: "a" | ... | "z" ;
<upper-case>  :: "A" | ... | "Z" ;
<decimal-digit> :: "0" | ... | "9" ;
<hex-digit>   :: <decimal-digit> | "A" | ... | "F" | "a" | ... | "f" ;
<simple-punc> :: "-" | "." | "/" | "_" | ":" | "*" | "+" | "=" ;
<whitespace-char> :: " " | "\t" | "\r" | "\n" ;
<base-64-char> :: <alpha> | <decimal-digit> | "+" | "/" | "=" ;
```

Wait! Did I say "fortunately"? This seems like a lot to digest, but going step by step, we can implement even a complex BNF such as this by converting each of these expressions to pyparsing elements. In fact, some of them are already built in to pyparsing, so we can just use them as is. Deep breath...OK, let's begin.

Since the published BNF is a "top-down" definition, we should work from the "bottom-up" in defining our pyparsing grammar. We need to do this since Python requires us to define elements before referencing them.

With some observation, we can see that the bottom half of this BNF consists mostly of defining sets of characters, rather than actual parsing expressions. In pyparsing, we will convert these definitions into strings that can be used in composing `Word` elements. This BNF also explicitly indicates whitespace in places. Since pyparsing skips over whitespace by default, we can leave these terms out where they are separators, and make sure that we accommodate whitespace when it is expressly part of an expression definition.

So, to begin expressing the sets of valid characters, let's review what pyparsing already provides us (a reminder, these are *strings*, not expressions, to be used in defining `Word` expressions):

`alphas`
: The characters A-Z and a-z

`nums`
: The characters 0–9

`alphanums`
: The combination of `alphas` and `nums`

`hexnums`

The combination of `nums` and the characters A-F and a-f

`printables`

All 7-bit ASCII characters that are not whitespace or control characters

If need be, we could also use the pyparsing function `srange` (for "string range"), which borrows the range syntax from regular expressions; for example, `srange("[A-Z]")` returns a string containing the uppercase letters A-Z.

We can now implement these basic character sets, working bottom-up, for composing our `Word` elements:

```
#<base-64-char> :: <alpha> | <decimal-digit> | "+" | "/" | "=" ;
base_64_char = alphanums + "+/="

#<whitespace-char> :: " " | "\t" | "\r" | "\n" ;
# not needed

#<simple-punc>       :: "-" | "." | "/" | "_" | ":" | "*" | "+" | "=" ;
simple_punc = "-./_:*+="

#<hex-digit>  :: <decimal-digit> | "A" | ... | "F" | "a" | ... | "f" ;
# use hexnums

#<decimal-digit> :: "0" | ... | "9" ;
# use nums

#<alpha>      :: <upper-case> | <lower-case> | <digit> ;
#<lower-case>        :: "a" | ... | "z" ;
#<upper-case> :: "A" | ... | "Z" ;
# use alphanums

#<token-char> :: <alpha> | <decimal-digit> | <simple-punc> ;
token_char = alphanums + simple_punc
```

Once again looking ahead at the next set of expressions, we can see that we will need some punctuation defined. The punctuation will be important during the parsing process, but I plan to convert the indicated fields during the parsing process using parse actions, so the punctuation itself will not be needed in the returned results. Using Python's list-to-variable assignment and the map function, we can define all our punctuation in a single compact statement:

```
LPAREN, RPAREN, LBRACK, RBRACK = map(Suppress, "()[]")
```

We can now visit the top half of the list, and implement those expressions that are defined in terms of the character sets and punctuation that have been defined:

```
# <bytes>     -- any string of bytes, of the indicated length
bytes = Word( printables )

# <decimal>   :: <decimal-digit>+ ;
#             -- decimal numbers should have no unnecessary leading zeros
decimal = "0" | Word( srange("[1-9]"), nums )

# <quoted-string-body> :: "\"" <bytes> "\""
# not needed, use dblQuotedString

# <quoted-string> :: <decimal>? <quoted-string-body>
quoted_string = Optional( decimal ) + dblQuotedString

# <hexadecimal>      :: "#" ( <hex-digit> | <white-space> )* "#" ;
hexadecimal = "#" + ZeroOrMore( Word(hexnums) ) + "#"

# <base-64>   :: <decimal>? "|" ( <base-64-char> | <whitespace> )* "|" ;
base_64 = Optional(decimal) + "|" + ZeroOrMore( Word( base_64_char ) ) + "|"

# <token>     :: <tokenchar>+ ;
token = Word( tokenchar )

# <raw>       :: <decimal> ":" <bytes> ;
raw = decimal + ":" + bytes

# <simple-string> :: <raw> | <token> | <base-64> | <hexadecimal> | <quoted-string> ;
simple_string = raw | token | base_64 | hexadecimal | quoted_string

# <display>   :: "[" <simple-string> "]" ;
display = LBRACK + simple_string + RBRACK

# <string>    :: <display>? <simple-string> ;
string_ = Optional(display) + simple_string
```

As I mentioned before, the published BNF describes elements in a general "top-down" order, or most complex expression to simplest expression. In developing these expressions in Python, we have to define the simple expressions first, so that they can be used in composing the more complicated expressions to follow.

Here are some other points in developing these next-level expressions:

- The definition of `decimal` states that there should be no extra leading zeros. We implemented this by using `Word`'s two-argument constructor, specifying the non-zero digits as the set of valid leading characters, and the set of all digits as the set of valid body characters.
- Most BNF's use the conventions of `+` for "1 or more," `*` for "zero or more," and `?` for "zero or one." For the most part, we can map these to pyparsing classes `OneOrMore`, `ZeroOrMore`, and `Optional`. (The exception is when the BNF indicates

`OneOrMore` of a set of characters; in this case, use the `Word` class, as shown with `token` being made up of `token_char+`.)

- This BNF includes optional whitespace as possible characters in the `hexadecimal` and `base-64` expressions. One way to implement this is to define an expression such as `hexadecimal` as `Word(hexnums+" ")`. Instead, I've chosen to define this as `ZeroOrMore(Word(hexnums))`, which will give us a list of the parts composed of hex digits, and implicitly skip over the interior whitespace.

At this point, the remaining expressions are:

```
<sexp>       :: <string> | <list>
<list>       :: "(" ( <sexp> | <whitespace> )* ")" ;
```

Here we have come to the recursive part of the BNF. In this case, `sexp` is defined in terms of `list`, but `list` is defined using `sexp`. To break this cycle, we'll define `sexp` as a `Forward` (and also rename `list` to `list_`, so as not to mask the Python built-in type):

```
# <list>     :: "(" ( <sexp> | <whitespace> )* ")" ;
sexp = Forward()
list_ = LPAREN + Group( ZeroOrMore( sexp ) ) + RPAREN
```

And to close the loop, we define the body of `sexp` as `(string_ | list_)`. Remember, we cannot just use a Python assignment statement or else the definition of `sexp` will not be used as the contents of `list_`. Instead, we use the `<<` shift operator to insert the definition into the previously defined `sexp`:

```
# <sexp>     :: <string> | <list>
sexp << ( string_ | list_ )
```

Now we have completed the basic syntax definition of the S-expression BNF. Let's try it out on our previous example, the authentication certificate:

Helpful Tip

To get nested output like this, I use Python's `pprint` pretty-printing module. You have to call the `asList`() method of the returned `ParseResults` to convert the `ParseResults` object into a simple nested list, which is understandable to the `pprint.pprint` method.

```
print sexp.parseString(certificateExpr)
```

gives us the results:

```
[['certificate',
  ['issuer',
```

```
    ['name',
     ['public-key',
      'rsa-with-md5',
      ['e','|', 'NFGq/E3wh9f4rJIQVXhS','|'],
      ['n',
       '|',
       'd738/4ghP9rFZ0gAIYZ5q9y6iskDJwAS
        i5rEQpEQq8ZyMZeIZzIAR2I5iGE=',
       '|']],
     'aid-committee']],
   ['subject',
    ['ref',
     ['public-key',
      'rsa-with-md5',
      ['e','|', 'NFGq/E3wh9f4rJIQVXhS','|'],
      ['n',
       '|',
       'd738/4ghP9rFZ0gAIYZ5q9y6iskDJwASi5rEQpEQq8ZyMZeIZzIAR2I5iGE=',
       '|']],
     'tom',
     'mother']],
   ['not-after', '"1998-01-01_09:00:00"'],
   ['tag',
    ['spend',
     ['account', '"12345678"'],
     ['*', 'numeric', 'range', '"1"', '"1000"']]]]]
```

Before leaving this parser, there are just a few more parse actions to add:

- Removal of the quotation marks on quoted strings.
- Conversion of the base64 data to binary data.
- Validation of the data length values, when specified.

The quotation marks on the quoted strings are simply there to enclose some body of text into a single string. The `dblQuotedString` expression already does this for us so that the leading and trailing quotation marks themselves are not needed in the results. This task is so common that pyparsing includes a method for just this purpose, `removeQuotes`:

```
dblQuotedString.setParseAction( removeQuotes )
```

To convert the base64 data field, we'll add a short parse action to call `b64decode`, defined in Python's `base64` module:

```
base_64 = Optional(decimal) + "|" + \
    OneOrMore(Word(base_64_char)).setParseAction(lambda t:b64decode("".join(t))) + "|"
```

This parse action does two tasks in one: it joins together the multiple tokens returned from the `OneOrMore` expression, and then calls `b64decode` to convert that data back to its original form.

Lastly, we'll implement the length validation for the `base-64`, `quoted-string`, and `raw` expressions. These expressions all include an optional leading `decimal` element to be used as an expected character count in the attached base64 or character string data. We can use a parse action to verify that these data length values are correct. We'll design the validation parse action to look for two named results in the input tokens: the length field will be named `length`, and the content field will be named `data`. Here is how that parse action will look:

```
def validateDataLength( tokens ):
    if tokens.length != "":
        if len(tokens.data) != int(tokens.length):
            raise ParseFatalException \
                ("invalid data length, %d specified, found %s (%d chars)" %
                 (int(tokens.length), tokens.data, len(tokens.data)))
```

At the start, `validateDataLength` checks to see whether a `length` field is given. If the `length` field is not present, retrieving the `length` attribute will return an empty string. If a `length` field is present, we then see whether the length of the `data` field matches it. If not, our parse action will a raise a different kind of `ParseException`, a `ParseFatalException`. Unlike the `Parse-Exception`, which simply signals a failed syntax match, `ParseFatal-Exception` will cause parsing to stop immediately at the current location.

For this parse action to work, we'll need to go back and attach results names to the appropriate elements of the `base_64`, `quoted_string`, and `raw` expressions:

```
raw = decimal.setResultsName("length") + ":" + Word(bytes).setResultsName("data")

dblQuotedString.setParseAction( removeQuotes )
quoted_string = Optional( decimal.setResultsName("length") ) + \
    dblQuotedString.setResultsName("data")

base_64_body = OneOrMore(Word(base_64_char))
base_64_body.setParseAction(lambda t:b64decode("".join(t)))
base_64 = Optional(decimal.setResultsName("length")) + "|" + \
    base_64_body.setResultsName("data") + "|"
```

The `base_64` expression was getting rather complicated, so I broke out the content field as its own expression, `base_64_body`.

With these changes in place, here is our final parser, with test cases:

```
from pyparsing import *
from base64 import b64decode
```

```
import pprint

LPAREN, RPAREN, LBRACK, RBRACK = map(Suppress, "()[]")

base_64_char = alphanums + "+/="
simple_punc = "-./_:*+="
token_char = alphanums + simple_punc
bytes = Word( printables )
decimal = ("0" | Word( srange("[1-9]"), nums )).setParseAction(lambda t: int(t[0]))

token = Word( token_char )
hexadecimal = "#" + ZeroOrMore( Word(hexnums) ) + "#"

dblQuotedString.setParseAction( removeQuotes )
quoted_string = Optional( decimal.setResultsName("length") ) + \
    dblQuotedString.setResultsName("data")

base_64_body = OneOrMore(Word(base_64_char))
base_64_body.setParseAction(lambda t:b64decode("".join(t)))
base_64 = Optional(decimal.setResultsName("length")) + \
    "|" + base_64_body.setResultsName("data") + "|"

raw = (decimal.setResultsName("length") + ":" +
        bytes.setResultsName("data"))
simple_string = raw | token | base_64 | hexadecimal | quoted_string
display = LBRACK + simple_string + RBRACK
string_ = Optional(display) + simple_string

sexp = Forward()
list_ = Group( LPAREN + ZeroOrMore( sexp ) + RPAREN )
sexp << ( string_ | list_ )

def validateDataLength( tokens ):
    if tokens.length != "":
        if len(tokens.data) != int(tokens.length):
            raise ParseFatalException \
                ("invalid data length, %d specified, found %s (%d chars)" %
                 (int(tokens.length), tokens.data, len(tokens.data)))

quoted_string.setParseAction( validateDataLength )
base_64.setParseAction( validateDataLength )
raw.setParseAction( validateDataLength )

######### Test data ###########
test0 = """(snicker "abc" (#03# |YWJj|))"""
test1 = """(certificate
 (issuer
  (name
   (public-key
    rsa-with-md5
```

```
    (e |NFGq/E3wh9f4rJIQVXhS|)
    (n |d738/4ghP9rFZ0gAIYZ5q9y6iskDJwASi5rEQpEQq8ZyMZeIZzIAR2I5iGE=|))
   aid-committee))
 (subject
  (ref
   (public-key
    rsa-with-md5
    (e |NFGq/E3wh9f4rJIQVXhS|)
    (n |d738/4ghP9rFZ0gAIYZ5q9y6iskDJwASi5rEQpEQq8ZyMZeIZzIAR2I5iGE=|))
   tom
   mother))
 (not-after "1998-01-01_09:00:00")
 (tag
  (spend (account "12345678") (* numeric range "1" "1000"))))
"""
test2 = """\
(defun factorial (x)
   (if (zerop x) 1
       (* x (factorial (- x 1)))))
       """
test3 = """(3:XX "abc" (#30# |YWJj|))"""

# Run tests
for t in (test0, test1, test2, test3):
    print '-'*50
    print t
    try:
        sexpr = sexp.parseString(t)
        pprint.pprint(sexpr.asList())
    except ParseFatalException, pfe:
        print "Error:", pfe.msg
        print line(pfe.loc,t)
        print pfe.markInputline()
    print
```

The test output is:

```
--------------------------------------------------
(snicker "abc" (#03# |YWJj|))
[['snicker', 'abc', ['#', '03', '#', '|', 'abc', '|']]]

--------------------------------------------------
(certificate
 (issuer
  (name
   (public-key
    rsa-with-md5
    (e |NFGq/E3wh9f4rJIQVXhS|)
    (n |d738/4ghP9rFZ0gAIYZ5q9y6iskDJwASi5rEQpEQq8ZyMZeIZzIAR2I5iGE=|))
   aid-committee))
 (subject
```

```
  (ref
   (public-key
    rsa-with-md5
    (e |NFGq/E3wh9f4rJIQVXhS|)
    (n |d738/4ghP9rFZ0gAIYZ5q9y6iskDJwASi5rEQpEQq8ZyMZeIZzIAR2I5iGE=|))
   tom
   mother))
 (not-after "1998-01-01_09:00:00")
 (tag
  (spend (account "12345678") (* numeric range "1" "1000"))))

[['certificate',
  ['issuer',
   ['name',
    ['public-key',
     'rsa-with-md5',
     ['e', '|', '4Q\xaa\xfcM\xf0\x87\xd7\xf8\xac\x92\x10UxR', '|'],
     ['n',
      '|',
     "w\xbd\xfc\xff\x88!?\xda\xc5gH\x00!\x86y\xab\xdc\xba\x8a\xc9\x03'\
       x00\x12\x8b\x9a\xc4B\x91\x10\xab\xc6r1\x97\x88g2\x00Gb9\x88a",
      '|']],
    'aid-committee']],
  ['subject',
   ['ref',
    ['public-key',
     'rsa-with-md5',
     ['e', '|', '4Q\xaa\xfcM\xf0\x87\xd7\xf8\xac\x92\x10UxR', '|'],
     ['n',
      '|',
     "w\xbd\xfc\xff\x88!?\xda\xc5gH\x00!\x86y\xab\xdc\xba\x8a\xc9\x03'\
       x00\x12\x8b\x9a\xc4B\x91\x10\xab\xc6r1\x97\x88g2\x00Gb9\x88a",
      '|']],
    'tom',
    'mother']],
  ['not-after', '1998-01-01_09:00:00'],
  ['tag',
   ['spend',
    ['account', '12345678'],
    ['*', 'numeric', 'range', '1', '1000']]]]]

--------------------------------------------------
(defun factorial (x)
   (if (zerop x) 1
       (* x (factorial (- x 1)))))

[['defun',
  'factorial',
  ['x'],
  ['if', ['zerop', 'x'], '1', ['*', 'x', ['factorial', ['-', 'x', '1']]]]]]
```

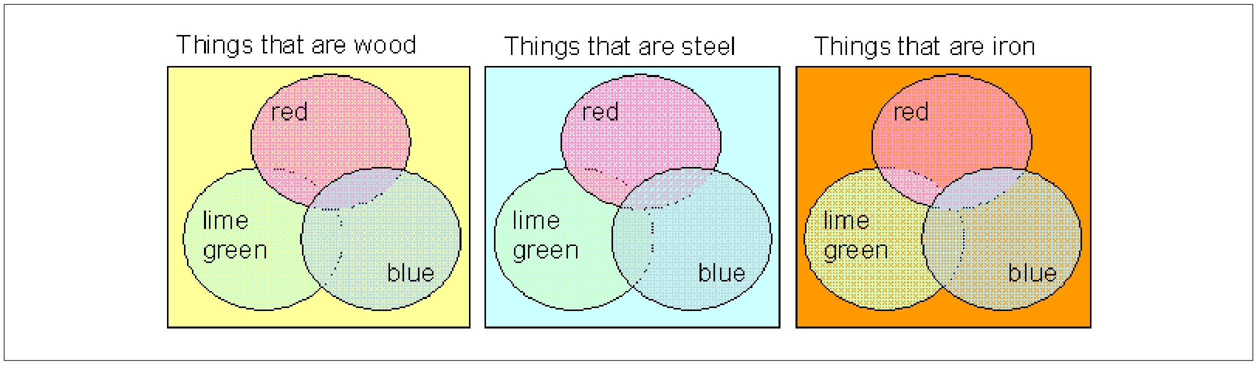


Figure 1. The universe of all possible things

```
-------------------------------------------------
(3:XX "abc" (#30# |YWJj|))
Error: invalid data length, 3 specified, found XX (2 chars)
(3:XX "abc" (#30# |YWJj|))
>!<
```

Parsing a Search String

The explosion of data made possible by the technologies of the Internet and the World Wide Web has led to the emergence of applications and services for searching and organizing that mass of data. A typical interface to a search service is a string of keywords that is used to retrieve web pages of interest to the searcher. Services such as Google have very simplified search interfaces, in which each separate word is assumed to be a potential keyword, and the search engine will look for pages containing any of the given keywords (perhaps ranking the pages by the number of keywords present on the page).

In this application, I am going to describe a more elaborate search string interface, with support for AND, OR, and NOT keyword qualifiers. Keywords may be single words delimited by whitespace, or a quoted string for keywords that contain spaces or non-alphanumeric characters, or for a search keyword or phrase that includes one of the special qualifier words AND, OR, or NOT. Here are a few sample search phrases for us to parse:

```
wood and blue or red
wood and (blue or red)
(steel or iron) and "lime green"
not steel or iron and "lime green"
not(steel or iron) and "lime green"
```

describing objects in the simple universe depicted in this figure.

We would also like to have our parser return the parsed results in a hierarchical structure based on the precedence of operations among the AND, OR, and NOT

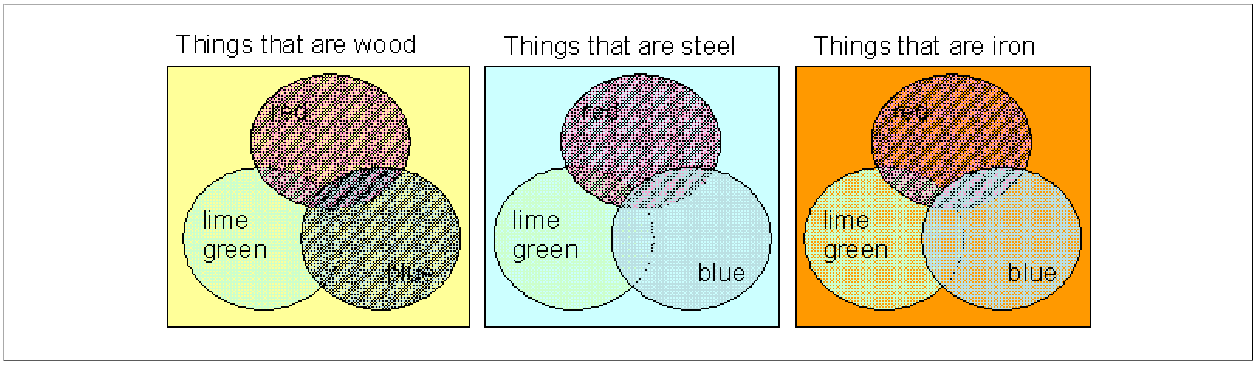


Figure 2. Things in the universe that are "wood and blue or red"

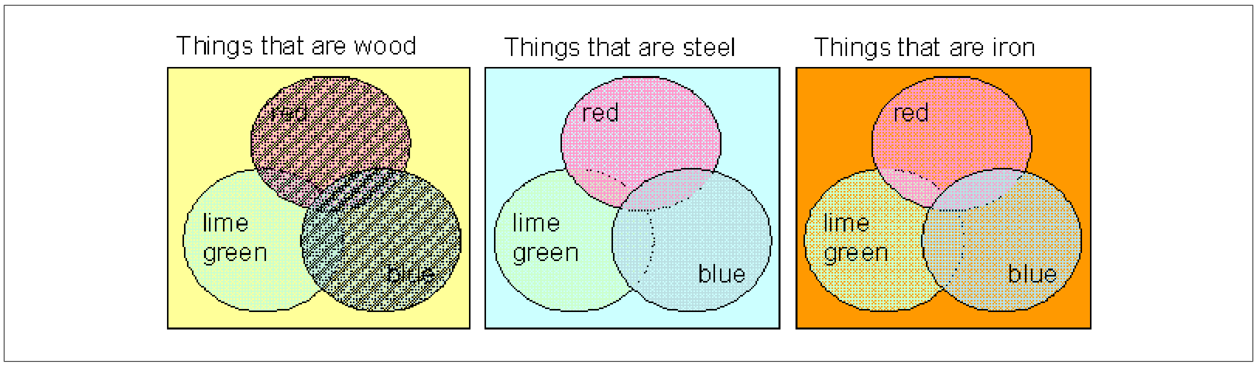


Figure 3. Things in the universe that are "wood and (blue or red)"

qualifiers. In normal programming practice, the hierarchy of these operations is defined as:

- NOT has the highest precedence, and is evaluated first
- AND has the next highest precedence
- OR has the lowest precedence, and is evaluated last

So, the phrase "wood and blue or red" is interpreted as "retrieve any item that is both wood and blue, OR any item that is red no matter what it is made of."

Parentheses can be used to override this precedence of operations, so that the phrase "wood and (blue or red)" will be interpreted as "only retrieve items that are made of wood AND are either blue or red."

Following our best practice, here is a BNF for this simple search string parser:

```
searchExpr ::= searchTerm [ ( AND | OR ) searchTerm ]...
searchTerm ::= [NOT] ( single-word | quotedString | '(' searchExpr ')' )
```

However, this BNF will not evaluate any precedence between AND and OR expressions. To do this, we must separate the AND and OR operations so that the precedence of evaluation will be represented in the parsed token structure.

To define a BNF that recognizes the precedence of operations, we must define a sequence of expressions in which each operation from lowest precedence to highest is parsed. The uppermost expression will define the operator with the lowest precedence, using as operands an expression that defines the operator with the next lowest precedence, and so on. At parsing time, the parser will recursively evaluate the expressions, so that through recursion, the highest precedence operator found in the input text will be recognized and evaluated first.

In this version of the BNF, AND expressions will be parsed ahead of OR expressions, and NOT expressions will be parsed ahead of AND expressions.

```
searchExpr ::= searchAnd [ OR searchAnd ]...
searchAnd  ::= searchTerm [ AND searchTerm ]...
searchTerm ::= [NOT] ( single-word | quotedString | '(' searchExpr ')' )
```

This grammar can be implemented directly into pyparsing expressions:

```
searchExpr = Forward()
searchTerm = Optional(not_) + ( Word(alphas) |
                                quotedString.setParseAction( removeQuotes ) | \
                                Group( LPAREN + searchExpr + RPAREN ) )
searchAnd = Group( searchTerm + ZeroOrMore( and_ + searchTerm ) )
searchExpr << Group( searchAnd + ZeroOrMore( or_ + searchAnd ) )
```

Using this grammar, our parsing examples evaluate as:

```
wood and blue or red
[[['wood', 'and', 'blue'], 'or', ['red']]]

wood and (blue or red)
[[['wood', 'and', [[['blue'], 'or', ['red']]]]]]

(steel or iron) and "lime green"
[[[[[['steel'], 'or', ['iron']]], 'and', 'lime green']]]

not steel or iron and "lime green"
[[['not', 'steel'], 'or', ['iron', 'and', 'lime green']]]

not(steel or iron) and "lime green"
[[['not', [[['steel'], 'or', ['iron']]], 'and', 'lime green']]]
```

These results can now be evaluated using a depth-first recursive method, and the AND, OR, and NOT expressions will be evaluated in the proper precedence.

Defining and evaluating expressions with operators of various precedence is a common application for parsers, and pyparsing includes a method named `operatorPrecedence` to simplify the definition of such grammars.

To use operatorPrecedence, you must first define a parse expression for the most basic operand. For this search application, the smallest operand is a search term given as a single word or quoted string:

```
searchTerm = Word(alphas) | quotedString.setParseAction( removeQuotes ) )
```

Using this base operand, you then compose the search expression by calling operatorPrecedence with this operand, and a list of operator descriptions. For each operator or set of operators with the same precedence level, you define:

- An expression to match the operator or set of operators
- The integer value 1 or 2 to indicate whether this is a unary or binary operator
- The pyparsing constant opAssoc.LEFT or opAssoc.RIGHT to describe whether this operator is left- or right-associative
- An optional parse action to be performed when an operation is matched

To define the NOT, AND, and OR operations for search terms, we can call operatorPrecedence using:

```
searchExpr = operatorPrecedence( searchTerm,
       [
       (not_, 1, opAssoc.RIGHT),
       (and_, 2, opAssoc.LEFT),
       (or_, 2, opAssoc.LEFT),
       ])
```

More Examples

The pyparsing wiki includes two other examples using operatorPrecedence:

- simpleBool: Evaluates Boolean logic expressions.
- simpleArith: Evaluates arithmetic expressions with the operations of addition, subtraction, multiplication, division, exponentiation, and factorial.

You can view them online, or find them in the examples directory of the pyparsing source and docs distributions.

Not only is this simpler to define, it also creates internal grouping, so that the test strings now parse to the simpler structures:

```
wood and blue or red
[[['wood', 'and', 'blue'], 'or', 'red']]


wood and (blue or red)
[['wood', 'and', ['blue', 'or', 'red']]]
```

```
(steel or iron) and "lime green"
[[['steel', 'or', 'iron'], 'and',
    'lime green']]

not steel or iron and "lime green"
[[['not', 'steel'], 'or', ['iron', 'and',
    'lime green']]]

not(steel or iron) and "lime green"
[[['not', ['steel', 'or', 'iron']], 'and',
    'lime green']]
```

Here is the complete parser code for our simple search expression parser:

```
from pyparsing import *

and_ = CaselessLiteral("and")
or_  = CaselessLiteral("or")
not_ = CaselessLiteral("not")
searchTerm = Word(alphanums) | quotedString.setParseAction( removeQuotes )
searchExpr = operatorPrecedence( searchTerm,
       [
       (not_, 1, opAssoc.RIGHT),
       (and_, 2, opAssoc.LEFT),
       (or_, 2, opAssoc.LEFT),
       ])

tests = """\
    wood and blue or red
    wood and (blue or red)
    (steel or iron) and "lime green"
    not steel or iron and "lime green"
    not(steel or iron) and "lime green"""".splitlines()

for t in tests:
    print t.strip()
    print searchExpr.parseString(t)[0]
    print
```

Results:

```
wood and blue or red
[['wood', 'and', 'blue'], 'or', 'red']

wood and (blue or red)
['wood', 'and', ['blue', 'or', 'red']]

(steel or iron) and "lime green"
[['steel', 'or', 'iron'], 'and', 'lime green']

not steel or iron and "lime green"
```

```
[['not', 'steel'], 'or', ['iron', 'and', 'lime green']]

not(steel or iron) and "lime green"
[['not', ['steel', 'or', 'iron']], 'and', 'lime green']
```

Search Engine in 100 Lines of Code

Let's build on the previous example and create an actual search engine. We can use parse actions to compile the search string into an intermediate data structure, and then use that structure to generate and evaluate Python sets to extract matching items by keyword.

The example data set this time will not be all the wooden, steel, or iron things in the universe that are red, blue, or lime green, but a collection of recipes with their respective ingredients.

These will be defined using simple Python lists:

```
recipes = "Tuna casserole/Hawaiian pizza/Chicken a la King/"\
    "Pepperoni pizza/Baked ham/Tuna melt/Eggs Benedict"\
        .split("/")
ingredients = "eggs/pineapple/pizza crust/pepperoni/ham/bacon/"\
    "English muffins/noodles/tuna/cream of mushroom soup/chicken/"\
    "mixed vegetables/cheese/tomato sauce/mayonnaise/Hollandaise sauce"\
        .split("/")
```

The contents of each recipe will be defined using a list of tuples, each tuple mapping the index of a recipe to the index of one of its ingredients:

```
recipe_ingredients_map = [
    (0,8),(0,9),(0,7),(1,2),(1,1),(1,4),(2,7),(2,9),(2,10),(2,11),
    (3,3),(3,2),(3,12),(3,13),(4,1),(4,4),(5,6),(5,8),(5,14),(5,12),
    (6,6),(6,0),(6,12),(6,4),(6,15),
    ]
```

So, recipe 0 ("Tuna casserole") contains ingredients 8, 9, and 7 ("tuna," "cream of mushroom soup," and "noodles"). Not exactly the Cordon Bleu, but enough to get us started.

Our search engine will focus on finding recipes that contain given ingredients, so we'll define a Python dict named `recipesByIngredient`, which will serve as our database index, to make it easier to find which recipes reference a particular ingredient:

```
recipesByIngredient = dict((i,[]) for i in ingredients)
for recIndex,ingIndex in recipe_ingredients_map:
    recipesByIngredient[ ingredients[ingIndex] ].append( recipes[recIndex] )
```

With our basic data reference established, we can work on applying the search string parser to extract data by search string.

The BNF for the search string is exactly as before, and we can reuse the `operatorPrecedence` implementation as is:

```
searchExpr ::= searchAnd [ OR searchAnd ]...
searchAnd  ::= searchTerm [ AND searchTerm ]...
searchTerm ::= [NOT] ( single-word | quotedString | '(' searchExpr ')' )

and_ = CaselessLiteral("and")
or_  = CaselessLiteral("or")
not_ = CaselessLiteral("not")
searchTerm = Word(alphanums) | quotedString.setParseAction( removeQuotes )
searchExpr = operatorPrecedence( searchTerm,
       [
       (not_, 1, opAssoc.RIGHT),
       (and_, 2, opAssoc.LEFT),
       (or_, 2, opAssoc.LEFT),
       ])
```

The next step is to modify the operation descriptions in the `operator-Precedence` definition to add parse actions to perform the creation of the search data structure. These parse actions will be different from previous parse actions we have created. Instead of modifying the string tokens or returning new string tokens, these parse actions will be class constructors that take a `ParseResults` object and *return an object* that will perform some form of data search evaluation. Each of the three operators NOT, AND, and OR will have its own search evaluation class, `SearchNot`, `SearchAnd`, and `SearchOr`.

Let's abstract the construction of these objects into two classes, `BinaryOperation` and `UnaryOperation`. The purpose of these classes will simply be to have an initializer that pulls the correct arguments from the tokens passed to a parse action, and saves them to suitable instance attributes:

```
class UnaryOperation(object):
    def __init__(self, tokens):
        self.op, self.a = tokens[0]
```

This simple initializer will extract the operator into an attribute named `op`, and the argument into attribute `a`.

```
class BinaryOperation(object):
    def __init__(self, t):
        self.op = tokens[0][1]
        self.operands = tokens[0][0::2]
```

The initializer for `BinaryOperation` looks a little more complicated. Binary operations in `operatorPrecedence` can return multiple operation/operands if multiple operations of the same precedence level occur together. For instance, a binary operation for addition will parse `"a+b+c"` as `[ 'a', '+', 'b', '+', 'c' ]`, not

[['a', '+', 'b'], '+', 'c'], so BinaryOperation needs to be able to recognize and process a chain of like operations, not just a simple a op b operation.

The Python [0::2] slicing notation indicates that the operands will be taken from the list of tokens beginning with the 0th element, and then stepping by 2 until the end of the list. So, this will process a token list such as ['a', '+', 'b', '+', 'c'], and give us the tokens ['a', 'b', 'c'].

For each operator type, we create a handler class that will derive from one of these classes. For now, let's just define handler classes that can display their own repr strings:

```
class SearchAnd(BinaryOperation):
    def __repr__(self):
        return "AND:(%s)" % (",".join(str(oper) for oper in self.operands))


class SearchOr(BinaryOperation):
    def __repr__(self):
        return "OR:(%s)" % (",".join(str(oper) for oper in self.operands))

class SearchNot(UnaryOperation):
    def __repr__(self):
        return "NOT:(%s)" % str(self.a)
```

To construct these objects during a parse action, you might consider creating parse actions that look something like:

```
def makeSearchAnd(tokens):
    searchAnd = SearchAnd(tokens)
    return searchAnd
...
```

and then pass makeSearchAnd, etc., into the operatorPrecedence call:

```
searchExpr = operatorPrecedence( searchTerm,
       [
       (not_, 1, opAssoc.RIGHT, makeSearchNot),
       (and_, 2, opAssoc.LEFT, makeSearchAnd),
       (or_, 2, opAssoc.LEFT, makeSearchOr),
       ])
```

In fact, the classes themselves are sufficient objects to pass as parse actions, since their initializers already have the proper argument lists to accept the arguments passed to a parse action. So, we can discard the makeSearchXXX methods, and just pass the classes directly as the parse actions for operatorPrecedence:

```
searchExpr = operatorPrecedence( searchTerm,
       [
       (not_, 1, opAssoc.RIGHT, SearchNot),
```

```
        (and_, 2, opAssoc.LEFT, SearchAnd),
        (or_, 2, opAssoc.LEFT, SearchOr),
        ])
```

For completeness, we will create one more class, to be added as a parse action for the base operand search terms themselves:

```
class SearchTerm(object):
    def __init__(self, tokens):
        self.term = tokens[0]
    def __repr__(self):
        return self.term

searchTerm.setParseAction( SearchTerm )
```

If we add these classes and the modified `operatorPrecedence` call to the existing parser code, we can visualize the search expressions data structures now being built. Once again, we will need some sample search strings to test the parser:

```
pineapple
pineapple and 'pizza crust'
noodles and tuna or ham
noodles and (tuna or ham)
pineapple and noodles
tuna or ham or pineapple
tuna or (ham and not pineapple)
not tuna
not (pineapple or tuna)
```

Since our returned objects implement suitable `repr` methods, we can just print the `ParseResults` object returned from calling `parseString`, and see the data structures that are created:

```
pineapple -> pineapple
pineapple and 'pizza crust' -> AND:(pineapple,pizza crust)
noodles and tuna or ham -> OR:(AND:(noodles,tuna),ham)
noodles and (tuna or ham) -> AND:(noodles,OR:(tuna,ham))
pineapple and noodles -> AND:(pineapple,noodles)
tuna or ham or pineapple -> OR:(tuna,ham,pineapple)
tuna or (ham and not pineapple) -> OR:(tuna,AND:(ham,NOT:(pineapple)))
not tuna -> NOT:(tuna)
not (pineapple or tuna) -> NOT:(OR:(pineapple,tuna))
```

Now that we are parsing the search string into a data structure of objects, we can put those objects to work searching for matching data.

The design we will implement will be very similar to the one used in the Venn diagrams that illustrated the search results matching the test strings in the first search string example. Each region in the diagram represents a set of objects. We will use Python sets to model these separate groups. For each `SearchXXX` class, we

will generate a fragment Python set expression, using Python set operations such as `&` for intersection, `|` for union, and `-` for negation. The concatenation of these fragments will give us an overall set expression, which can be evaluated using the Python `eval` command. The results of the evaluated set expression will be the desired results of the original search string. We will in essence have compiled the search string into a Python expression, which we then execute to get the requested search results.

`SearchTerm` is the simplest to implement and will set the stage for the more complex operations. When a term such as "pineapple" is included in the search string, we would like to select all of the recipes that include "pineapple" as one of their ingredients. Given the in-memory mini-database we created at the beginning of this section, we can find the set of all recipes directly from the global dict variable `recipesByIngredient`:

```
set( recipesByIngredient['pineapple'] )
```

We should also guard against searches for ingredients that are not in the database so that if a term is not in `recipesByIngredient`, we return an empty set.

So, the implementation of `generateSetExpression` for `SearchTerm` is:

```
    def generateSetExpression(self):
        if self.term in recipesByIngredient:
            return "set(recipesByIngredient['%s'])" % self.term
        else:
            return "set()"
```

`SearchAnd` and `SearchOr` are almost as easy. Since these binary operations have an `operands` attribute initialized to contain the terms or subexpressions that are to be "and"ed or "or"ed, the `generateSetExpression` method for `SearchAnd` and `SearchOr` can be the results of the operands' `generateSetExpression` methods, joined by `&` and `|`, respectively:

```
# class SearchAnd
    def generateSetExpression(self):
        return "(%s)" % \
                  " & ".join(oper.generateSetExpression() for oper in self.operands)

# class SearchOr
    def generateSetExpression(self):
        return "(%s)" % \
                  " | ".join(oper.generateSetExpression() for oper in self.operands)
```

`SearchNot` is a bit more difficult. The set expression for the set of items that are not in set X is the set of all items minus X. In Python, we would implement ~X as:

```
set(everything) - set(X)
```

To implement `generateSetExpression` for `SearchNot`, we will return the negation of the operand's set expression as the operand's set subtracted from the set of all recipes:

```
# class SearchNot
    def generateSetExpression(self):
        return "(set(recipes) - %s)" % self.a.generateSetExpression()
```

That completes the work on the search classes. Let's rerun the tests to see how the generated set expressions look:

```
pineapple ->
     set(recipesByIngredient['pineapple'])

pineapple and 'pizza crust' ->
     (set(recipesByIngredient['pineapple']) & set(recipesByIngredient['pizza crust']))

noodles and tuna or ham ->
     ((set(recipesByIngredient['noodles']) & set(recipesByIngredient['tuna']))
| set(recipesByIngredient['ham']))

noodles and (tuna or ham) ->
     (set(recipesByIngredient['noodles']) & (set(recipesByIngredient['tuna']) |
set(recipesByIngredient['ham'])))

pineapple and noodles ->
     (set(recipesByIngredient['pineapple']) & set(recipesByIngredient['noodles']))

tuna or ham or pineapple ->
     (set(recipesByIngredient['tuna']) | set(recipesByIngredient['ham']) |
set(recipesByIngredient['pineapple']))

tuna or (ham and not pineapple) ->
     (set(recipesByIngredient['tuna']) | (set(recipesByIngredient['ham']) &
(set(recipes) - set(recipesByIngredient['pineapple']))))

not tuna ->
     (set(recipes) - set(recipesByIngredient['tuna']))

not (pineapple or tuna) ->
     (set(recipes) - (set(recipesByIngredient['pineapple']) |
set(recipesByIngredient['tuna'])))

anchovies ->
     set()
```

(I added the last search string as a test of the nonexistent ingredient search.)

The only remaining step for each search string is to evaluate its Python set expression, and list out the resulting items. The following code lists the complete parser, including test data initialization and test search strings:

```
from pyparsing import *

# populate ingredients->recipes "database"
recipes = "Tuna casserole/Hawaiian pizza/Chicken a la King/"\
    "Pepperoni pizza/Baked ham/Tuna melt/Eggs Benedict"\
        .split("/")
ingredients = "eggs/pineapple/pizza crust/pepperoni/ham/bacon/"\
    "English muffins/noodles/tuna/cream of mushroom soup/chicken/"\
    "mixed vegetables/cheese/tomato sauce/mayonnaise/Hollandaise sauce"\
        .split("/")
recipe_ingredients_map = [
    (0,8),(0,9),(0,7),(1,2),(1,1),(1,4),(2,7),(2,9),(2,10),(2,11),
    (3,3),(3,2),(3,12),(3,13),(4,1),(4,4),(5,6),(5,8),(5,14),(5,12),
    (6,6),(6,0),(6,12),(6,4),(6,15),
    ]
recipesByIngredient = dict((i,[]) for i in ingredients)
for recIndex,ingIndex in recipe_ingredients_map:
    recipesByIngredient[ ingredients[ingIndex] ].append( recipes[recIndex] )

# classes to be constructed at parse time, from intermediate ParseResults
class UnaryOperation(object):
    def __init__(self, t):
        self.op, self.a = t[0]

class BinaryOperation(object):
    def __init__(self, t):
        self.op = t[0][1]
        self.operands = t[0][0::2]

class SearchAnd(BinaryOperation):
    def generateSetExpression(self):
        return "(%s)" % \
                  " & ".join(oper.generateSetExpression() for oper in self.operands)
    def __repr__(self):
        return "AND:(%s)" % (",".join(str(oper) for oper in self.operands))

class SearchOr(BinaryOperation):
    def generateSetExpression(self):
        return "(%s)" % \
                  " | ".join(oper.generateSetExpression() for oper in self.operands)
    def __repr__(self):
        return "OR:(%s)" % (",".join(str(oper) for oper in self.operands))

class SearchNot(UnaryOperation):
    def generateSetExpression(self):
        return "(set(recipes) - %s)" % self.a.generateSetExpression()
```

```
    def __repr__(self):
        return "NOT:(%s)" % str(self.a)

class SearchTerm(object):
    def __init__(self, tokens):
        self.term = tokens[0]
    def generateSetExpression(self):
        if self.term in recipesByIngredient:
            return "set(recipesByIngredient['%s'])" % self.term
        else:
            return "set()"
    def __repr__(self):
        return self.term

# define the grammar
and_ = CaselessLiteral("and")
or_  = CaselessLiteral("or")
not_ = CaselessLiteral("not")
searchTerm = Word(alphas) | quotedString.setParseAction( removeQuotes )
searchTerm.setParseAction(SearchTerm)
searchExpr = operatorPrecedence( searchTerm,
       [
       (not_, 1, opAssoc.RIGHT, SearchNot),
       (and_, 2, opAssoc.LEFT, SearchAnd),
       (or_, 2, opAssoc.LEFT, SearchOr),
       ])

# test the grammar and selection logic
test = """\
    pineapple
    pineapple and 'pizza crust'
    noodles and tuna or ham
    noodles and (tuna or ham)
    pineapple and noodles
    tuna or ham or pineapple
    tuna or (ham and not pineapple)
    not tuna
    not (pineapple or tuna)
    anchovies""".splitlines()

for t in test:
    try:
        evalStack = (searchExpr + stringEnd).parseString(t)[0]
    except ParseException, pe:
        print "Invalid search string"
        continue
    print "Search string:", t
    # print "Eval stack:    ", evalStack
    evalExpr = evalStack.generateSetExpression()
    # print "Eval expr:    ", evalExpr
```

```
    matchingRecipes = eval(evalExpr)
    if matchingRecipes:
        for r in matchingRecipes: print "-", r
    else:
        print "  (none)"
    print
```

And here are the test results:

```
Search string:     pineapple
- Baked ham
- Hawaiian pizza

Search string:     pineapple and 'pizza crust'
- Hawaiian pizza

Search string:     noodles and tuna or ham
- Baked ham
- Eggs Benedict
- Hawaiian pizza
- Tuna casserole

Search string:     noodles and (tuna or ham)
- Tuna casserole

Search string:     pineapple and noodles
  (none)

Search string:     tuna or ham or pineapple
- Eggs Benedict
- Tuna melt
- Hawaiian pizza
- Tuna casserole
- Baked ham

Search string:     tuna or (ham and not pineapple)
- Eggs Benedict
- Tuna melt
- Tuna casserole

Search string:     not tuna
- Pepperoni pizza
- Baked ham
- Eggs Benedict
- Hawaiian pizza
- Chicken a la King

Search string:     not (pineapple or tuna)
- Pepperoni pizza
- Eggs Benedict
- Chicken a la King
```

```
Search string:    anchovies
  (none)
```

At the end, we find that we have created a complete search engine in less than 100 lines of code—time to go chase some venture capital!

Conclusion

When I gave my presentation on pyparsing at PyCon '06, one of the questions after my talk was, "Is there anything you *can't* do with pyparsing?" This may have been a response to some of my posts to comp.lang.python, in which I recommended using pyparsing in many nontraditional applications, and often as an alternative to using regular expressions. I stammered a bit, and I mentioned that pyparsing is not always the best-suited tool— some data is already pretty well structured, and is better parsed using string indexing and `str.split()`. I also do not recommend pyparsing for processing XML—there are already parsing and data access utilities out there, and applications that need XML typically need better performance than pyparsing will deliver.

But I think pyparsing is an excellent tool for developing command processors, web page scrapers, and parsers of text datafiles (such as logfiles or analysis output files). Pyparsing has been embedded in several popular Python add-on modules; go to the pyparsing wiki (***http://pyparsing.wikispaces.com/whosusingpyparsing***) for links to the latest ones.

I've written some documentation for pyparsing, but I have probably spent more time developing code examples that demonstrate various pyparsing code techniques. My experience has been that many developers want to see a selection of source code, and then adapt it to their problem at hand. Recently, I've started getting email asking for more formal usage documentation, so I hope this Short Cut helps those who want to get going with pyparsing.

For More Help

There are a growing number of online resources for pyparsing users:

- Pyparsing wiki (***http://pyparsing.wikispaces.com***): This wiki is the primary source for all news about pyparsing. It includes download information, FAQs, and links to projects that are using pyparsing for their integrated parsers. An Examples page gives a variety of "how to" cases, including parsers for arithmetic expressions, chess game notation, JSON data, and SQL statements (these are also included in the pyparsing source distribution). Pyparsing releases, presentations, and other events are posted on the News page. And the discussion threads on the main home page are an easy-to-use resource for posting questions or browsing the points raised by other pyparsing users.
- Pyparsing mailing list (*pyparsing-users@lists.sourceforge.net*): Another general resource for posting pyparsing questions. An archive of previous list messages can be found at the pyparsing SourceForge project page, ***http://sourceforge.net/projects/pyparsing***.
- comp.lang.python: This Usenet group is a general-purpose Python discussion group, but an occasional pyparsing-related thread will crop up here. This is a good group for posting questions related to Python usage, or to locate specialized modules for a particular application or field of study. If you Google "pyparsing" in this list, you will find a number of archived threads on a variety of applications.

Index